ffice of
cademic
omputing

IBM PROGRAMMING SUPPORT PACKAGES FOR NSW

NSW Semi-Annual Technical Reports

January 1, 1977 - June 30, 1977
and
July 1, 1977 - December 31, 1977

UCLA TR-22

Neil Ludlam
Robert Braden
Lou Rivas
Denis De La Roca

niversity of
alifornia,
os
ngeles

87 10 14 138

SECURITY CLASSIFICATION OF THIS

REPORT DOC

AD-A186 802

READ INSTRUCTIONS
BEFORE COMPLETING FORM

1. REPORT NUMBER
OAC/TR22

RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
IBM Programming Support Packages
National Software Works

TYPE OF REPORT & PERIOD COVERED
Semi-Annual Tech Reports
1/1/77 - 12/31/77

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
R. Braden N. Ludlam
D. DeLa Roca L. Rivas

8. CONTRACT OR GRANT NUMBER(s)
MDA 903-74-C-0083

9. PERFORMING ORGANIZATION NAME AND ADDRESS
University of California
Office of Academic Computing
Los Angeles, CA 90024

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
Prog. Elemt: 62708E
Program Code: 0T10
ARPA Order No. 2543

11. CONTROLLING OFFICE NAME AND ADDRESS
Defense Advanced Research Projects Agency
1440 Wilson Blvd., Arlington, VA 22209
Attn: Program Mgt. Office

12. REPORT DATE
11/15/80

13. NUMBER OF PAGES
176

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)
Defense Supply Service - Washington
Room 10-245, The Pentagon
Washington, D.C. 20310
R. Mueller

15. SECURITY CLASS. (of this report)

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Distribution Unlimited

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
National Software Works, ARPANET, protocol, NSW, exchange mechanism, PL/I interfaces, PLOXI, PL/MSG, interprocess communications, MSG, PLIB8 data encodement, NTP, NSWB8, PL/PCP, procedure calls, NTP, programming discipline, PLIDAIR, dynamic allocation, TXSTAX, attention handling

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
This report covers technical development at UCLA relating to the National Software Works (NSW) during 1977. It is specifically concerned with the design and development of subroutine packages necessary to implement the system interface of the various levels of the NSW Transaction Protocol (NSWTP) under OS/MVT.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

IBM PROGRAMMING SUPPORT PACKAGES FOR NSW

by
Neil Ludlam
Robert Braden
Louis Rivas
Denis De La Roca

November 15, 1980

Document TR-22

UCLA Office of Academic Computing
5628 Math Sciences Addition
University of California C0012
Los Angeles, California 90024

This work was sponsored by
the Advanced Research Projects Agency
of the Department of Defense,
under ARPA Order no. 2543,
Contract No. MDA 903-74-C-0083:

ARPANET COMPUTER SERVICES IN SUPPORT OF
THE NATIONAL SOFTWARE WORKS

June 1, 1975 - February 29, 1980

William B. Kehl, Principal Investigator
(213) 825-7511

SEMI-ANNUAL TECHNICAL REPORTS
for period of
January 1, 1977 - December 31, 1977

A

REPORT SUMMARY

This report covers technical development at UCLA relating to the National Software Works (NSW) during 1977. It is a combination of the two Semi-annual Technical Reports covering the periods of January 1 through June 30 and July 1 through December 31 of 1977.

The primary goal of the NSW project at UCLA is to make the IBM Operating System OS/MVT, and specifically its implementation on the UCLA IBM 360/91, a "tool-bearing host" within the NSW. This report is specifically concerned with the subroutine packages that we have developed to implement system interfaces and the various levels of the NSW Transaction Protocol (NSWTP) under OS/MVT. Not only do these packages ease design and programming, but they normalize the interface between a program on the UCLA machine that is acting as an NSW process and the remote components of NSW, and they localize future maintenance activities.

Subsequent sections of the report document specific subroutine packages. Each section corresponds to a document stored in the NSW documentation repository maintained by the NSW Operations Contractor, so each section has been made self-contained. For example, each section has its own table of contents and reference summary, and each section is independently paginated.

Part II: PLOXI

This section describes the PLOXI subroutine package, which provides a general-purpose interface between any program written in PL/I (for the IBM Optimizing Compiler) and UCLA's "Exchange" mechanism. Exchange is the only general-purpose inter-process communication mechanism available to a program running under OS/MVT. It is used for all communication between an NSW program and MSG, the process responsible for communicating with other NSW hosts. PLOXI is particularly important for the implementation of MSG "direct connections" such as those that connect the NSW Front End component to an executing interactive tool, or those used by the NSW File Package components to transfer data among themselves.

Part III: PL/MSG

This section describes the PL/MSG subroutine package, which implements the MSG primitive-operation interface for OS/MVT. This section appeared in a slightly less complete form in UCLA Technical Report TR-12, the NSW Semi-annual report for the period from July 1 to December 31, 1976. It is repeated here to incorporate significant changes, and also to make this report a comprehensive manual of NSW protocol subroutine packages as implemented for OS/MVT.

Part IV: PL/B8

This section describes the PL/B8 subroutine package, which implements the interface to the data-encodement aspect of the NSW Network Transaction Protocol (NTP), known as NSWB8. This encodement is used for all messages transmitted between NSW processes via MSG, excepting MSG direct connections, which always use private protocols. PL/B8 translates data between NSWB8 and a form that is appropriate for use in a PL/I program.

Part V: PL/PCP

This section describes the PL/PCP subroutine package, which implements an interface to the procedure-call aspect of the NSW Network Transaction Protocol (NTP). While NTP does not itself formalize the procedure-call constructs implemented by PL/PCP, a higher-level protocol is implicit in the NTP specification. PL/PCP explicates that protocol, standardizes it, and localizes changes that will be required by the expected embellishments in future versions of NTP. Also, the programming discipline imposed by PL/PCP has been a definite aid in NSW program design and implementation at UCLA.

Part VI: PLIDAIR

This section describes the PLIDAIR subroutine package, which implements an interface to OS/MVT's Dynamic Allocation Interface Routine (DAIR). This interface allows a PL/I program to perform create, delete, rename, etc. operations on local disk data sets. Such operations are, of course, vital to the operation of a monitor-like system such as NSW.

Part VII: TXSTAX

This section describes the TXSTAX subroutine package, which implements an interface between an operating PL/I program and the attention-interruption handling mechansims of OS/MVT. Attention handling is necessary in order to support various test vehicles that we will have to use to exercise NSW.

In OS/MVT, a time-sharing job which is waiting for terminal input is totally and irrevocably blocked, not only in the waiting task, but in all tasks of the job. For this reason, it is virtually impossible to write a program which monitors NSW communications activity and is receptive to commands from a controlling console. Unfortunately, this is exactly the type of program that is needed for almost any form of NSW testing or monitoring.

For these reasons, we will have to implement such programs to focus their attention on the NSW, attending to the controlling console only in response to an attention interruption. The vehicle for monitoring for this interruption is the TXSTAX package.

PART II

PLOXI -- A PL/I INTERFACE TO EXCHANGE

This section is separately available
as UCLA document UCNSW-407

2. PART II: THE PLOXI PACKAGE

2.1. INTRODUCTION

PLOXI is a package of assembly-language subroutines designed to be called by PL/I (IBM optimizing compiler) programs wishing to access UCLA's Exchange. Through Exchange, the PL/I program can communicate with another job, with another task of the same job, with another routine of the same task, or with any system task which can provide a matching Exchange request.

PLOXI was inspired by PLEXI, an Exchange interface package for F-level PL/I written by Vic Tolmei (reference 2). This document is an update of reference 3. For a more complete explanation of the Exchange, see reference 1.

PLOXI includes the following entry points:

EXOPEN begins Exchange communication by requesting an "open window". Parameters to EXOPEN include symbolic names which can match corresponding names in another EXOPEN issued (for example) by a program in another job. When such a match occurs, a window is opened by Exchange.

EXCH transfers data through an open Exchange window. When this call matches an EXCH issued by the program on the other side of the window, data transfer occurs and the EXCH requests "complete" (except in STREAM mode -- see reference 1).

EXCLOSE closes a particular Exchange window.

EXWAIT waits for completion of EXOPEN and/or EXCH requests.

EXTEST tests for completion of EXOPEN and/or EXCH requests.

In general, EXCH and EXOPEN operations are asynchronous to the calling PL/I program. For example, suppose the programmer opens an Exchange window and issues an EXCH in the beginning of his program. That EXCH may remain pending for an unpredictable time before completing. On the other hand, the EXCH may complete immediately, before the EXCH subroutine returns to the PL/I program, because it matches a pending EXCH previously issued by the other program.

The PLOXI routines EXOPEN, EXCLOSE, and EXCH basically build new parameter lists and issue the corresponding Exchange SVC's. PLOXI itself supplies the necessary parameter areas (EXB's) for all EXCH's which can be pending at once. The PLOXI EXOPEN subroutine issues a GETMAIN and sets up a pool of EXB's for this purpose. Each EXCH call allocates an EXB from the pool, returning to the caller a "handle" for it called a "sequence id". The PL/I program can subsequently pass

this to EXWAIT, which blocks the task (via an OS WAIT SVC) until the operation completes. When the operation does complete, either EXCH or EXWAIT will return the EXB to the free pool.

Each EXOPEN or EXCH parameter area includes an Event Control Block (ECB). When the corresponding operation completes, Exchange posts this ECB with a "post code" indicating success or failure in the match (and consequent data transfer in the case of EXCH).

A call to the EXCH subroutine includes a "type" parameter specifying whether or not the program will continue execution while the EXCH is pending. The choices are:

Type='WAIT': execution of the PL/I task will be blocked until the EXCH completes. At that time, the ECB will have been posted.

Type='POST': PL/I execution will continue after the EXCH call. When the EXCH does complete, the ECB will be posted. The program must later call the EXTEST or EXWAIT subroutine to determine the status and complete the data transfer.

Type='FAIL': the EXCH will complete normally (with data transferred) only if it can complete immediately. If it would have remained pending under type 'POST', it instead fails immediately and posts an error code in the ECB.

2.2. EXOPEN -- OPEN AN EXCHANGE WINDOW

This subroutine issues the *EXOPEN SVC* to request an Exchange connection or "window". The request may require an immediate match (notify='FAIL') or may remain pending until a matching EXOPEN SVC is executed by another program. Control may return at once to PL/I (notify='POST'), or be suspended until the window opens (notify='WAIT').

Unless there is an error in its call, the EXOPEN subroutine obtains an area of core in the PL/I region for a "window control block" (XWCB) and a pool of EXB's, and returns a "handle" to the XWCB called an "openid".

2.2.1. DECLARATION

```
DECLARE EXOPEN
            ENTRY (CHAR(4),
                   *,
                   FIXED BIN(31),
                   FIXED BIN(31))
                       OPTIONS (ASSEMBLER, INTER);
```

2.2.2. CALLING SEQUENCE

```
CALL EXOPEN (notify,
             window,
             openid,
             retcode);
```

Where:

"notify" (CHAR(4)) indicates the mode of notification of the PL/I program. It contains one of the strings: 'POST', 'WAIT', or 'FAIL'.

"window" (structure, as defined below) contains the static description of the attributes of the window, including symbolic tags to be matched with another open request.

"openid" (FIXED BIN(31)) is an integer returned by EXOPEN. If the request is erroneous, this value will be zero; otherwise, it will be an integer equal to the address of a new Window Control Block (XWCB). This integer must be used by the PL/I program as a handle to identify this window in subsequent calls to EXCH, EXTEST, EXWAIT, and EXCLOSE.

"retcode" (FIXED BIN(31)) is an integer return code set by EXOPEN. Its values are given below. A text message is also moved into the window structure.

2.2.3. WINDOW STRUCTURE

```
DECLARE
  1 WINDOW,
    2 WINDOW_ID     FIXED BIN(31),
    2 XWCBPTR       POINTER,
*   2 CHANNUMB      FIXED BIN(31),
*   2 MAXPERCHAN    FIXED BIN(31),
*   2 TRACELIM      FIXED BIN(31),
*   2 YOURTAG       CHAR (8),
*   2 MYTAG         CHAR (8),
*   2 YOURJOB       CHAR (8),
*   2 MYJOB         CHAR (8),
    2 RETMSG        CHAR (16),
    2 TRACEAREA     (tracelim/16) CHAR(16);
```

The call to EXOPEN must include a window structure of this form, with the starred entries filled in. The entries have the following meanings:

WINDOW_ID is a number filled in by EXOPEN which uniquely identifies the open window. It is not normally needed by the PLI program.

XWCBPTR is a pointer to the XWCB. This is the same value as "openid". It is placed here by EXOPEN as a possible convenience to the PL/I program.

CHANNUMB is the number of independent "channels" to be provided in the window.

MAXPERCHAN is the maximum number of EXCH's per channel which may be pending at one time.

TRACELIM is the size of the optional TRACEAREA entry at the end of the WINDOW structure. It must be a multiple of 16. If it is zero, no trace area is provided.

YOURTAG initially specifies the required symbolic tag of the other side, or '*' to indicate that the caller wants to connect with any tag. When the EXOPEN completes, YOURTAG will contain the actual MYTAG of the other side, as CHAR(8).

MYTAG specifies a symbolic tag for the local side of the window. It cannot be blank or '*'.

YOURTAG initially specifies the required symbolic name of the job on the other side, or blanks to indicate the name of the job issuing the EXOPEN, or '*' to indicate that the caller wants to connect with any job. When the EXOPEN completes, YOURJOB will contain the actual MYJOB of the other side, as CHAR(8).

MYJOB should normally be a BLANK string, to imply the jobname of the job issuing the EXOPEN. Privileged jobs can specify a non-blank MYJOB.

RETMSG is a text message expanding on the return code from the last call to any of the PLOXI routines for this window. Its values are enumerated below along with those of the return code.

TRACEAREA (optional) is a vector of "n" 16-byte trace entries from the last "n" operations, with the oldest entry first. The total length n*16 must be the value of TRACELIM.

2.2.4. RETURN CODES AND MESSAGES

0/'OPEN WINDOW' -- The Window opened successfully. OPENID is non-zero and YOURTAG, YOURJOB, WINDOW_ID, and XWCBPTR are all set.

4/'PENDING' -- Notify='POST' was specified, and the request is pending. Openid and XWCBPTR are set. EXWAIT or EXTEST must be called with 'openid' as a handle until the window opens, to complete the window structure.

40016/'FAILED' -- Notify='FAIL' was specified, and no match was found. Openid has been set to zero and no XWCB remains from this call.

80000/'BADxxxx...' -- The request failed because of bad parameters. Here "xxxx...." specifies the faulty parameter, or is blank if the error was detected by the EXOPEN SVC. Openid has been set to zero and no XWCB remains from this call.

2.2.5. ABENDS

It is possible for the EXOPEN argument list to be so erroneous that EXOPEN cannot locate a "retcode" variable. In this case, the subroutine will issue its "retmsg" as a WTP and abend with user code 1001. Possible causes are:

1) There are too few or too many arguments in the call.

2) The retcode argument is not fullword aligned.

3) The window structure is not fullword aligned.

4) The Exchange is out of service (return code 20 from the SVC).

2.3. EXCH -- ISSUE AN EXCHANGE REQUEST

This subroutine begins data transmission and signalling by issuing an EXCH SVC. This request will generally complete when a matching EXCH is issued by the other side.

2.3.1. DECLARATION

This routine has two forms of declaration, depending on whether it is to use "locate" mode on the one hand, or "move" or "stream" mode on the other. For locate mode, the declaration is:

```
DECLARE EXCH
            ENTRY (CHAR(6),
                   FIXED BIN(31),
                   FIXED BIN(31),
                   FIXED BIN(31),
                   FIXED BIN(31),
                   FIXED BIN(31),
                   FIXED BIN(31))
                       OPTIONS (ASSEMBLER, INTER);
```

For move or stream mode, the declaration is:

```
DECLARE EXCH
            ENTRY (CHAR(6),
                   FIXED BIN(31),
                   FIXED BIN(31),
                   CHAR (*) VAR,
                   CHAR (*) VAR,
                   FIXED BIN(31),
                   FIXED BIN(31))
                       OPTIONS (ASSEMBLER, INTER);
```

In fact, combinations of the modes are permitted, with the obvious changes in the declarations.

2.3.2. CALLING SEQUENCE

```
CALL EXCH (typemode,
           openid,
           channel,
           msgstring,
           repstring,
           exchid,
           retcode);
```

Where:

"typemode" (CHAR(6)) specifies the type of EXCH request, and -- when relevant -- the data transfer modes. Valid strings are:

'POSTmr': Completion will be signalled by posting the ECB. EXWAIT must be called to wait on it, and/or EXTEST must be called to test it.

'WAITmr': The calling task blocks until the EXCH completes. EXCH calls EXWAIT internally, so no explicit call to EXWAIT is necessary.

'FAILmr': The operation is to be done only if it can be completed synchronously and immediately.

'TEST': The call is to test for pending (queued) EXCH's but is not to match any (cause them to complete) or to become queued itself. Completion results in the integer count of EXCH's queued on the channel being returned as part of "retcode".

'CANCEL': The call is to cancel the first pending EXCH issued by this program. That EXCH will complete with a return code indicating cancellation without data transfer.

'RESET': The call is to cancel all pending EXCH's issued by this program. Those EXCH's will complete with a return code indicating cancellation without data transfer.

In the first three types above, the letters "m" and "r" stand for single characters which specify the message and reply data transfer modes, respectively. Either may be one of the following:

'L' means locate mode.

'M' means move mode.

'S' means stream mode.

'*' means no transfer in this direction.

"openid" (FIXED BIN(31)) is the handle returned from EXOPEN.

"channel" (FIXED BIN(31)) is the channel number to be used. Channels are numbered {0, 1, ..., CHANNUMB-1}.

"msgstring" (FIXED BIN(31)) in locate mode, CHAR(*) VAR otherwise) is the message to be sent. This operand is ignored if "typemode" is one of:

'tttt*r'
'TEST'
'CANCEL'
'RESET'

"repstring" (FIXED BIN(31)) in locate mode, CHAR(*) VAR otherwise) is the reply buffer. The EXCH operation may shorten a varying repstring if less than the maximum possible data is sent, but it will never lengthen it. That is, the length of repstring when EXCH is called determines how much data can be received in this EXCH operation. This operand is ignored if "typemode" is one of:

'ttttm*'
'TEST'
'CANCEL'
'RESET'

"exchid" (FIXED BIN(31)) is an integer handle returned by EXCH to be used as a "seqid" in a later call to EXTEST or EXWAIT. It is set to zero if the request fails due to an error, or if "typemode" is one of:

'TEST'
'CANCEL'
'RESET'

"retcode" (FIXED BIN(31)) is an integer return code set by EXCH. Its values are given below. If a trace area is contained in the WINDOW structure, then a text message is also moved into that area.

2.3.3. RETURN CODES -- DATA TRANSFER MODES

The retcode values from EXCH are derived from the ECB post code. The general rules for this derivation are:

If the ECB is not posted, then retcode = 4.

Else, if ECB = X'4rxxyynn' then retcode = r*10000 + max(xxyy,nn)

Generally speaking, there are three cases to be considered for EXCH type='POST', 'WAIT', or 'FAIL' return codes:

Retcode=0 -- EXCH has completed fully, and the EXB (internal PLOXI control block) has been freed. It is redundant to call EXWAIT, but it is possible to do so; EXWAIT will ignore the ECB and return 8 in this case. However, if EXWAIT is to be called, this MUST be done before another EXCH call against the same window, because that would possibly reuse the freed EXB.

Retcode=4 -- EXCH is pending (only possible for type='POST'). EXWAIT or EXTEST must be called using "exchid" as a handle until that routine returns a value other than 4. The "exchid" is actually the address of the PLOXI EXB used to hold the parameters for the pending EXCH.

Retcode>4 -- An error was detected and the request was ignored. "Exchid" has been set to zero. In the following list of such error codes, those marked with "S" are synchronous errors. That is, these errors will always be detected immediately by EXCH and will cause failure of the operation. The codes marked with "A" may be asynchronous. That is, they may be detected by EXCH, or later by EXTEST or EXWAIT.

* Expected exceptional conditions

  40004 (A) -- other side closed window or abended.

  40008 (A) -- completed by 'CANCEL'.

  40012 (A) -- completed by 'RESET'.

  40016 (S) -- type='FAIL' failed.

* Programmer errors

  80004 (S) -- MAXPERCHAN exceeded.

  80012 (S) -- invalid "typemode".

  80016 (S) -- invalid channel number.

  80020 (S) -- invalid or closed window.

  80024 (S) -- Exchange failed catastrophically.

* Data transfer errors

12xxyy (A) -- A data transfer error has occurred. "xx" is the message error code, and "yy" is the reply error code. Either can be:

00: no error.

04: other side had a data transfer error.

08: mismatched modes.

16: STREAM mode restart error (caller's extent list clobbered since beginning of data transfer).

20: message or reply address was bad.

24: bad extent list on indeterminate side (Exchange got a program interrupt trying to move data).

120012 (A) -- deadlock detected

120028 (S) -- other side's EXB is clobbered.

2.3.4. RETURN CODES -- CONTROL MODES

An EXCH of type='TEST', 'CANCEL', or 'RESET' always completes immediately unless there is an error, so it always returns an "exchid" of zero. Possible return codes are:

0 -- Type='CANCEL' completed normally.

nnn1 -- Type='RESET' or 'TEST': "nnn" gives the integer count of pending exchanges. If it is less than 256 it is the number of exchanges pending on the other side. If it is 256 or greater, then nnn-256 is the number of exchanges pending on this side of the window.

40004 -- other side closed the window or abended.

40008 -- type='CANCEL': no exchanges were pending.

80004 -- MAXPERCHAN exceeded.

80012 -- invalid "typemode".

80016 -- invalid channel number.

80020 -- invalid window id, or window is closed.

80024 -- Exchange failed catastrophically.

2.3.5. ABENDS

One of these EXCH errors will cause a WTP of an error message and an ABEND with user completion code 1003:

1) Wrong number of arguments.

2) Fullword argument not on a fullword boundary.

3) SVC return code of 16 (clobbered control blocks) or 20 (Exchange not operational).

2.4. EXWAIT -- WAIT FOR EVENTS TO COMPLETE

Basically, this subroutine issues an OS WAIT SVC for a list of ECB's signifying completion of specified EXOPEN requests, EXCH requests and/or other events. Execution will be suspended until at least one of these events completes. The caller can then determine which event completed by testing the EXWAIT return codes.

For those requests which do complete, the EXWAIT routine also finishes the operation (EXOPEN or EXCH) as if it had completed when originally issued. For example, in the case of an EXOPEN request, it fills in YOURTAG and YOURJOB; in the case of EXCH it completes the data transfer and returns the EXB to the free pool.

2.4.1. DECLARATION

```
DECLARE EXWAIT
            ENTRY (FIXED BIN(31),
                   (*) FIXED BIN(31),
                   (*) FIXED BIN(31))
                       OPTIONS (ASSEMBLER, INTER);
```

or

```
DECLARE EXWAIT
            ENTRY (FIXED BIN(31),
            FIXED BIN(31))
                       OPTIONS (ASSEMBLER, INTER);
```

2.4.2. CALLING SEQUENCE

```
CALL EXWAIT (n,
             seqid-list,
             retcode-list);
```

or

```
CALL EXWAIT (seqid,
             retcode);
```

Where:

"n" (FIXED BIN(31)) is the number of (seqid, retcode) pairs to be included in the wait list.

"seqid-list" ((n) FIXED BIN(31)) is a vector of "n" "seqid" values (see "seqid").

"seqid" (FIXED BIN(31)) is an integer handle which identifies the event whose completion is to be awaited. It may b :

1) An "openid", to await completion of an EXOPEN request.

2) An "exchid", to await completion of an EXCH request.

3) The 2's complement of the address of a non-PLOXI ECB, to wait on an arbitrary event.

4) Zero, if a simple placeholder is needed; such an event never completes, and its corresponding "retcode" always receives 8.

"retcode-list" ((n) FIXED BIN(31)) is a vector of "n" words to receive "retcode" values (see "retcode").

"retcode" (FIXED BIN(31)) is an integer return code set by EXWAIT. Its values are given below.

2.4.3. RETURN CODES

In general, a return code of 4 or 8 indicates that the corresponding event is still pending. Any other code indicates completion of that event, but the specific meaning depends upon the type of event.

* Bad or Null Control Blocks or Bad Call

One of the following return codes will be given if the control block(s) pointed to by the corresponding seqid failed the PLOXI validity checks.

8 -- The normal return code for a null (zero) seqid.

80028 -- The seqid is incorrect; it points neither to a recognizable XWCB nor to an EXB which points to an XWCB. Either the seqid, the XWCB, or the EXB area is clobbered.

* EXOPEN request

The only two possible return codes are:

0 -- The window has opened and the WINDOW structure has been completed exactly as if the original EXOPEN had returned 0. In particular, RETMSG is set to "OPEN WINDOW". A subsequent call to EXWAIT or EXTEST will also return 0.

4 -- The open is still pending. This is possible only if "n" > 1.

* EXCH request

The following return codes are possible:

0 -- The EXCH request has been completed exactly as if the original EXCH had completed normally. The corresponding "seqid" (=exchid) variable has been set to zero.

4 -- The EXCH is still pending. This is possible only if "n" > 1.

40004 -- The other side closed the window or abended.

40008 -- The EXCH was completed by 'CANCEL'.

40012 -- The EXCH was completed by 'RESET'.

12xxyy -- There was a data transfer error (see this code under EXCH).

* Other ECB's

A non-PLOXI ECB may be included in the list by setting its "seqid" argument to the 2's complement of the pointer to (address of) the ECB. This may require some "UNSPECing" in PL/I.

0 -- The event has completed.

4 -- The event is still pending. This is possible only if "n" > 1.

2.4.4. ABENDS

One of these errors will cause a WTP of an error message and an ABEND with user completion code 1002:

1) There are more than three arguments.

2) An argument is not on a fullword boundary.

2.5. EXTEST -- TEST FOR EVENTS TO COMPLETE

This subroutine is an alternative entry point to EXWAIT. It functions just as does EXWAIT except that the actual OS WAIT is bypassed. EXTEST simply computes return codes and finishes completed requests.

2.5.1. DECLARATION

```
DECLARE EXTEST
            ENTRY (FIXED BIN(31),
                   (*) FIXED BIN(31),
                   (*) FIXED BIN(31))
                       OPTIONS (ASSEMBLER, INTER);
```

or

```
DECLARE EXTEST
            ENTRY (FIXED BIN(31),
            FIXED BIN(31))
                       OPTIONS (ASSEMBLER, INTER);
```

2.5.2. CALLING SEQUENCE

```
CALL EXTEST (n,
             seqid-list,
             retcode-list);
```

or

```
CALL EXTEST (seqid,
             retcode);
```

Where the arguments are exactly as defined for EXWAIT.

2.5.3. RETURN CODES

The return codes from EXTEST are the same as those from EXWAIT; however, while a code of 4 can be returned from EXWAIT only if "n" is greater than 1, that code is commonly returned from EXTEST in any call.

2.5.4. ABENDS

One of these errors will cause a WTP of an error message and an ABEND with user completion code 1002:

1) Wrong number of arguments (not 2 or 3).

2) An argument is not on a fullword boundary.

3) "n" > 20.

2.6. EXCLOSE -- CLOSE EXCHANGE WINDOW(S)

This subroutine issues an EXCLOSE SVC to close either a specified list of open windows or else all open windows on this task. In the first case, EXCLOSE also frees the parameter area(s) obtained by EXOPEN.

2.6.1. DECLARATION

```
DECLARE EXCLOSE
            ENTRY (FIXED BIN(31),
                   (*) FIXED BIN(31),
                   (*) FIXED BIN(31))
                       OPTIONS (ASSEMBLER, INTER);
```

or

```
DECLARE EXCLOSE
            ENTRY (FIXED BIN(31),
                   FIXED BIN(31))
                       OPTIONS (ASSEMBLER, INTER);
```

or

```
DECLARE EXCLOSE
            ENTRY (FIXED BIN(31))
                       OPTIONS (ASSEMBLER, INTER);
```

2.6.2. CALLING SEQUENCE

```
CALL EXCLOSE (n,
              openid-list,
              retcode-list);
```

or

```
CALL EXCLOSE (openid,
              retcode);
```

or

```
CALL EXCLOSE (0);
```

Where:

"n" (FIXED BIN(31)) is the number of (openid, retcode) pairs, i.e., the number of windows to be closed. In the special case of n=0, all open windows on this task will be closed, whether they were opened by PLOXI or not; however, the PLOXI control-block areas will not be freed for any of these windows.

"openid-list" ((n) FIXED BIN(31)) is a vector of "n" "openid" values (see "openid").

"openid" (FIXED BIN(31)) is an integer handle as returned by EXOPEN.

"retcode-list" ((n) FIXED BIN(31)) is a vector of "n" words to receive "retcode" values (see "retcode").

"retcode" (FIXED BIN(31)) is an integer return code set by EXCLOSE. Its values are given below.

2.6.3. RETURN CODES

Only two return codes are defined:

0 -- All requested windows are closed.

40000 -- At least one openid is bad (e.g., the window is already closed.

2.6.4. ABENDS

One of these errors will cause a WTP of an error message and an ABEND with user completion code 1005:

1) Wrong number of arguments (more than 3).

2) An argument is not on a fullword boundary.

3) "n" > 20.

REFERENCES

[1] Tolmei, "PLEXI -- a PL/I to Exchange Interface". UCLA document S-173, April 4, 1975.

[2] Braden and Feigin, "Programmer's Guide to the Exchange". UCLA document TR-5, March, 1972.

[3] Braden, "A PLI (Optimizer) Interface to Exchange". UCLA document S-191, February 29, 1976.

PART III

PL/MSG -- AN MSG INTERFACE FOR PL/I

This section is separately available
as UCLA document UCNSW-401

3. PART III: THE PL/MSG PACKAGE

3.1. INTRODUCTION

PL/MSG is a set of subroutines which invoke the functions of UCLA's implementation of the National Software Works' (NSW) communication facility MSG (reference 1). It is assumed here that the reader is familiar with that facility. MSGBUG is the name given to a debugging and control interface to PL/MSG. This interface is also described in this document.

In some of the discussions to follow, it will be clarifying to understand the relationship between PL/MSG and MSG itself. MSG is a protocol consisting of a set of "primitives" as defined by reference 1. At UCLA, MSG is also a subprocess of the UCLA Network Control Program (NCP) which manages this protocol in a manner of no concern to the PL/MSG user. PL/MSG, on the other hand, is a package consisting of subroutine calls which cause MSG primitives to be executed. Some PL/MSG calls also perform non-primitive MSG functions, or supporting functions which are not explicitly defined by the MSG protocol. Thus we will speak of "issuing MSG primitives" or of "calling PL/MSG routines."

Both the MSG section of the UCLA NCP and the PL/MSG interface were designed and written by Lou Rivas of UCLA. The MSGBUG debugging interface to PL/MSG was designed and written by Neil Ludlam of UCLA. These programs and their documentation are in the public domain. This document is a revision of sections 2 and 3 of reference 5.

3.2. BASIC PL/MSG CALLS

There are PL/MSG calls for MSG primitive operations and for some non-primitive MSG operations. The parameters used by these calls are selected from the data structures described in the section PL/MSG DATA STRUCTURES. The values returned in event signals are described in the section entitled EXCEPTIONAL CONDITIONS. Actual functions of MSG primitives are described in the NSW MSG documentation (reference 1). Only information not covered in one of these places is included in this section.

The PL/MSG routines are described in terms of their invocation from PL/I (IBM Optimizing Compiler) programs; however, those in this section can be invoked from any language that can supply the data structures described. In order to avoid PL/I dependencies, these routines do not accept PL/I data descriptors (dope vectors). This requires that the PL/I programmer declare these entries with the attribute

```
OPTIONS (ASSEMBLER, INTER)
```

but this will be done automatically if he uses the %INCLUDE segments named after the PL/MSG routines to declare their entry attributes. See the section entitled CANNED PL/I DECLARATIONS.

3.2.1. MATERIALIZING A PROCESS (TERMINATIONSIGNAL)

```
CALL MSGMP  (process name,     /* Set and returned */
             process handle,   /* PL/MSG returns   */
             event signal);    /* PL/MSG posts     */
```

This non-primitive operation materializes your process, introduces you to MSG, assigns you a specific process name, and provides you with an event signal corresponding to the MSG terminationsignal primitive. This is the only form in which that primitive is available in this implementation.

"Process name" should be your chosen generic name, with all fields except "generic_name" zero. PL/MSG returns values for the other fields, yielding your specific process name. It also returns a handle to the process in "process handle", but you need not remember or use this value unless you plan to materialize multiple MSG processes within a single program. See the section entitled MATERIALIZING MULTIPLE PROCESSES.

In the future, we expect to allow a specific form of "process name", corresponding to an attempt by an aborted process to restart itself and re-establish interrupted communication with MSG. Definition of this case is deferred.

3.2.2. RECEIVEGENERICMESSAGE

```
CALL MSGRGM  (message area,     /* Set and returned */
              process name,     /* PL/MSG returns   */
              event signal,     /* PL/MSG posts     */
              timeout interval); /* You must set    */
```

You must give a maximum value to the length field of "message area". PL/MSG will set a true length when it fills in the message data, and will also return a specific process name.

3.2.3. RECEIVESPECIFICMESSAGE

```
CALL MSGRSM  (message area,     /* Set and returned */
              process name,     /* PL/MSG returns   */
              event signal,     /* PL/MSG posts     */
              timeout interval, /* You must set     */
              special handling); /* PL/MSG returns  */
```

You must give a maximum value to the length field of "message area". PL/MSG will set a true length when it fills in the message data, and will also return a specific process name and values for "special handling".

3.2.4. RECEIVEALARM (ENABLEALARM)

MSG documentation refers to this primitive as "enablealarm". We will call it "receivealarm" in UCLA documentation; the original name has been found to cause confusion.

```
CALL MSGRA (alarm code,      /* PL/MSG returns   */
            process name,    /* PL/MSG returns   */
            event signal,    /* PL/MSG posts     */
            timeout interval); /* You must set   */
```

PL/MSG will fill in "alarm code" and all parts of "process name".

3.2.5. SENDGENERICMESSAGE

```
CALL MSGSGM  (message area,     /* You must set     */
              process name,     /* You must set     */
              event signal,     /* PL/MSG posts     */
              timeout interval, /* You must set     */
              wait enable);     /* You must set     */
```

You must give values to "message area", "process name", and "wait enable". "Process name" must be a generic name, with zeros in all fields except "generic_name", and optionally, "host_number".

3.2.6. SENDSPECIFICMESSAGE

```
CALL MSGSSM  (message area,      /* You must set     */
              process name,      /* You must set     */
              event signal,      /* PL/MSG posts     */
              timeout interval,  /* You must set     */
              special handling); /* You must set     */
```

You must give values to "message area", "process name", and "special handling". "Process name" must be a specific name, with all fields filled in.

3.2.7. SENDALARM

```
CALL MSGSA  (alarm code,         /* You must set     */
             process name,       /* You must set     */
             event signal,       /* PL/MSG posts     */
             timeout interval);  /* You must set     */
```

You must give values to "alarm code" and all parts of "process name".

3.2.8. RESCIND

```
CALL MSGRSND (event signal); /* You must set */
```

In this special case, "event signal" is used to identify the pending event to be rescinded. It is not associated with the completion of the rescind primitive, which does not itself create a pending event. In this implementation, it will be signalled and set to indicate that the event with which it was originally associated was aborted by a rescind. However, if it is found to be already signalled complete, (there will be a race here) this call becomes a no-operation. Notice that the ADDRESS of "event signal" is actually used to locate the original event -- so passing a copy will not work.

3.2.9. ARMALARMS (ACCEPTALARMS)

MSG documentation refers to this primitive as "acceptalarms". We will call it "armalarms" in UCLA documentation, as the original name has been found to cause confusion.

```
CALL MSGAA (alarm arm); /* You must set */
```

where "alarm arm" is '1'B if you wish to arm the process for alarms, and '0'B if you wish to disarm.

3.2.10. RESYNC

```
CALL MSGRSNC (process name); /* You must set */
```

where "process name" must be a specific name, with all fields filled in.

3.2.11. STOPME

```
CALL MSGSTOP;
```

In the UCLA implementation, this call dematerializes the current MSG process. It does not affect program or task status in any other way. On return, the calling program is free to materialize another process, terminate, or continue execution dealing with non-MSG work.

3.3. MANAGING DIRECT CONNECTIONS

There is a separate set of PL/MSG calls for manipulating MSG direct connections. These differ primarily in that they are written in PL/I, and can be called by PL/I-compiled programs only. No OPTIONS attribute should be declared for these entries. This difference will be transparent to the PL/I programmer if he uses the %INCLUDE segments named after the PL/MSG routines to declare their entry attributes. See the section entitled CANNED PL/I DECLARATIONS.

3.3.1. TCAM CONNECTIONS

In the UCLA implementation, there is a special variant of the Server Telnet (STEL) connection called the TCAM connection. This variant is defined for the purpose of encapsulating the terminal communications of programs running in native mode under TSO, and diverting that terminal activity into an STEL connection. The primary differences between a TCAM and an STEL connection are:

1) The *program* opening the connection must specify 'TCAM', not 'STEL' as the connection type.

2) The openconn will fail if the program is running under a real TSO session with a real terminal attached. It can only succeed if the "terminal" is in fact the MSG process-creation mechanism.

3) However, if the MSGBUG "NOTCAM" order is in effect, the openconn will succeed, but it will be converted internally into an ordinary STEL connection. Normally, this will only allow the openconn to succeed -- since the two types of connection use different methods of data transfer, it is unlikely that the program will be prepared to actually transfer data on an STEL connection.

4) Data cannot be read or written on the TCAM connection using MSGGET or MSGPUT. The OS macros TGET and TPUT are used instead. These are the macros that are used by ordinary TSO command processors.

3.3.2. OPENING A DIRECT CONNECTION (OPENCONN)

In the UCLA implementation, the openconn and closeconn primitives are implemented in a way that makes them technically non-primitive operations. The PL/MSG caller need be aware of this only in that these events can only be revoked by closing the connection, not

through the rescind primitive.

```
CALL MSGOC (connection request,/* You must set      */
            process name,      /* You must set      */
            timeout interval,  /* You must set      */
            connection handle, /* PL/MSG returns    */
            open event signal, /* PL/MSG posts      */
            close event signal); /* PL/MSG posts    */
```

You are responsible for ensuring that the "connection identifier" subfield of "connection request" is known to the remote process, which should specify it in its matching openconn request. On return from this call the value of the connection handle will be set, whether the event is complete or not. You should save this handle as your means of referencing the connection in subsequent PL/MSG calls.

The "open event signal" will be posted when the connection has opened successfully. The "close event signal" will be posted whenever the connection is closed, either:

* due to a failure to open in the first place;

* due to a closeconn issued from this process;

* due to a closeconn issued from the remote process;

* due to an unrecoverable failure in the network communications machinery between the two processes.

In any case, if it is ever posted before this process has issued a closeconn, you should:

* Dispose of all pending GETs and PUTs against the window (all will have completed in one way or another, and GETs may have provided you with good data).

* Issue closeconn.

3.3.3. CLOSING A DIRECT CONNECTION (CLOSECONN)

```
CALL MSGCC (connection handle); /* You must set */
```

The connection is identified by the connection handle returned by openconn. In addition to the MSG-specified functions of the closeconn primitive, in the UCLA implementation, this call can be used to rescind connection related events -- that is, you need not await completion of a pending MSGOC, MSGPUT, or MSGGET before issuing this call, if your intention is to abort the event. Because of the particularities of the UCLA implementation, completion of this operation is signalled through the close ECB that was passed to PL/MSG at openconn time. If that ECB was posted before this call,

then this call will complete immediately (though that does not mean that it is superfluous). The ECB will not be cleared or posted again, so you can still wait on it if that is convenient.

In order to avoid lost data, two processes sharing a connection must usually synchronize their use of closeconn by means of the protocols used for communication on the connection. PL/MSG recommends these sequences to minimize such dependence on higher level protocols:

* Case 1 -- abort: You can close any type of connection at any time, if your intention is to abort the associated activities, just by calling MSGCC.

* Case 2 -- responding to a close: any time a connections's close event signal is posted, implying that the other process has initiated closeconn, you should clean up and issue closeconn as soon as practical. This is the recomended technique in the case of binary simplex receive connections.

* Case 3 -- binary simplex send: for this type of connection, closeconn can be issued at any time without loss of data.

* Case 4 -- binary duplex: to initiate closeconn, use MSGEOD to "close" the send half of the connection. This has the effect of draining your send buffers and posting the other process' close event signal. That process is then expected to close the connection, draining your input buffers and posting your close event signal. You are then expected to call MSGCC.

* Case 5 -- TELNET types: These conversational connections require some higher-level-protocol synchronization. The only general rule is that the UTEL side should normally initiate the scenario.

3.3.4. GETTING DATA FROM A DIRECT CONNECTION

```
CALL MSGGET (connection handle,/* You must set      */
             message area,      /* Set and returned */
             event signal);     /* PL/MSG posts     */
```

You must have set a maximum length in "message area". PL/MSG will return a buffer full of data from the connection identified by "connection handle", and will adjust the length accordingly. The PL/MSG connection handler is insensitive to the transmission units, if any, used by the processes sharing a direct connection, and treats the data as a single logical stream. The amount of data returned from a single MSGGET depends on the connection type:

* Case 1 -- TELNET types: MSGGET returns the least of the length you supply, the amount that has arrived from the other process, or data up to and including either a "newline" or a "goahead" character. It will only wait if nothing at all has arrived from the remote process. This technique means that, theoretically, you can receive input one character at a time; however, given the operational characteristics of the ARPANET, actual operation will tend to be in units of terminal lines. In the current implementation data may be lost if the buffer is less than 256 bytes.

  The normal EBCDIC codes for "newline" and "goahead" are hex '15 and' 'BFB9', respectively. However, if you have arranged special options from the UCLA telnet protocol handlers, these codes may appear in other forms. The mechanism for setting options in the telnet protocol is independent of MSG, and is documented elsewhere.

* Case 2 -- Binary types: MSGGET will not complete until the message area has been filled, or until it becomes aware that the remote process has initiated closeconn.

MSGGET always returns an integral number of 8-bit SYSTEM/360 bytes. If the connection bytesize is not a multiple of 8, it is the responsibility of the caller of MSGGET to interpret the buffer as a free bit string, and to locate true byte boundaries. Excess bits at the end of one such string must be saved and concatenated ahead of the next string received. Excess bits following the very last complete transmission byte are defined to be implementation-required padding, and may be discarded. In the special case of a transmission bytesize of less than 8 bits, it may not be possible to determine how many bits of the final 8-bit SYSTEM/360 byte are significant. For this reason the use of connection bytesizes of less than 8 bits is discouraged in this implementation.

3.3.5. SENDING DATA OVER A DIRECT CONNECTION

```
CALL MSGPUT (connection handle,/* You must set     */
             message area,     /* You must set     */
             event singal);    /* PL/MSG posts     */
```

You must provide output data in "message area," and must not alter it until the event signal is posted. MSGPUT always accepts an integral number of 8-bit SYSTEM/360 bytes. If the connection bytesize is not a multiple of 8, it is the responsibility of the caller of MSGPUT to concatenate transmission bytes into a free bit string for MSGPUT. Bits that overreach the last SYSTEM/360 byte boundary must be saved and concatenated ahead of the next string to be transmitted. The very last SYSTEM/360 byte must be padded by the caller, if necessary.

When MSGCC or MSGEOD is called, PL/MSG recognizes that excess bits following the very last complete transmission byte are implementation-required padding, and may be discarded. In the special case of a transmission bytesize of less than 8 bits, it may not be possible for PL/MSG to determine how many bits are insignificant, resulting in one or more meaningless padding bytes being transmitted. For this reason the use of connection bytesizes of less than 8 bits is discouraged in this implementation.

3.3.6. MSGEOD -- PREPARING FOR CLOSECONN

```
CALL MSGEOD (connection handle);  /* You must set  */
```

MSGEOD is a preliminary to MSGCC. It can be issued against any connection, but it only has a specific function in the case of type FULL. It indicates that you intend to issue no more puts against the connection; however, you may still issue gets. See the reccommended usage of MSGEOD under CLOSING A DIRECT CONNECTION, above.

3.4. SUPPLEMENTARY SUBROUTINES

These routines are not MSG primitives, and are not absolutely necessary to use PL/MSG; however, they do add convenience.

3.4.1. MSGWAIT -- WAITING FOR EVENTS

```
CALL MSGWAIT (event signal, ...);
```

This routine is provided as a programming convenience. Its parameter list consists of any number of event signal names. Control does not return until at least one of the named event signals is posted.

3.4.2. MSGHTYP -- CLASSIFYING AN NSW HOST

```
CALL MSGHTYP (host number,  /* You must set      */
              host name,    /* PL/MSG returns    */
              host family); /* PL/MSG returns    */
```

This routine accepts an NSW host number and returns character strings containing human-oriented names for the host and for the NSW host family of which it is a part. These strings contain no variable information, so the outputs of two calls to MSGHTYP can be compared to determine whether two hosts are in the same family.

3.4.3. MSGJOUR -- WRITING TO THE PROCESS JOURNAL

```
CALL MSGJOUR (message,       /* You must set      */
              message type); /* You must set      */
```

This routine accepts an character string of up to 256 bytes, determines if it is of a type that is being journalled, and if so, writes it to the currently-enabled journal destinations. The values of "message type" are taken from %INCLUDE segment MSGJOUR, which is listed in an appendix.

3.4.4. MSGSETP -- USING SPECIAL POST SCHEDULING

```
CALL MSGSETP (routine addr,  /* You must set      */
              bit mask);     /* You must set      */
```

This routine is provided as an aid to the construction of higher-level protocol support packages using PL/MSG. It is not expected to be useful outside that context. The routine address is a PL/I POINTER variable which contains the address of an ASSEMBLER entry point, and the bit mask is a filter that associates the entry point with only certain MSG operations. The bits of the mask are numbered 1 - 16, which numbers correspond to the values of the primitive number field of the MSG event signal (see the section entitled EVENT SIGNALS). For every primitive which corresponds to a

'1' bit in the mask, the routine will be established as that primitive's posting routine.

A posting routine receives control with registers 0 and 1 set up exactly as for the OS POST macro. Registers 14 and 15 are as used by BALR. Register 13 points to a useable save area. No other registers may be assumed to have known values, and no other registers may be altered when the routine returns. The routine is executed on an IRB, so it MUST be a closed subroutine, and it MUST NOT use PL/I.

The posting status of a primitive with a 0-bit in the call's mask is not changed. Thus the effects of multiple calls are cumulative.

The second parameter may be omitted, in which case a default of all 1's will be used. If the pointer is NULL, a default internal routine will be used. That routine merely posts the event signal in the usual way. Thus, a call with a NULL pointer can be used to undo the effects of a previous call.

3.5. PL/MSG DATA ELEMENTS

This section describes various data elements that are used for communication between a caller and the PL/MSG routines. Each is described in terms of a PL/I data declaration, and its values are discussed in general terms. Any details on the use of a data structure in the context of a particular PL/MSG call are given in the section entitled BASIC PL/MSG CALLS.

3.5.1. PROCESS NAMES

A process name is a structure of the form:

```
DCL 1 process_name,
      2 host_number       FIXED BIN(15),
      2 host_incarnation  FIXED BIN(15),
      2 process_instance  FIXED BIN(15),
      2 generic_name      CHAR(127) VAR;
```

where for a generic name, only "generic_name" need have a non-null value. "host_number" may have the value assigned one of the NSW host computers, or may be zero to indicate that any host will do. The other two data must be zero for a generic name.

For a specific name, all values must be non-zero. The calling program will have gotten the values that are not used in a generic name from MSG (though sometimes through another program), and will only be concerned with passing them unchanged in subsequent PL/MSG calls.

When you are receiving a process name from PL/MSG it is important that "generic name" be declared CHAR(127) VARYING. When you are providing the datum to PL/MSG, this string need only be long enough to accommodate its value, but it must be VARYING.

3.5.2. PROCESS HANDLES

A process handle is a datum of the form:

```
DCL process_handle POINTER;
```

It is generated by PL/MSG at process materialization time, and may be used by the calling program as a process handle. However, the caller need not manage this datum unless he is materializing multiple concurrent processes within the same program. See the section entitled MATERIALIZING MULTIPLE PROCESSES for more information.

3.5.3. EVENT SIGNALS

A PL/MSG event signal is a datum of the form:

```
DCL event_signal FIXED BIN(31);
```

It is used as an OS/360 Event Control Block (ECB), and it is useful to redeclare it in the following form:

```
DCL 1 ecb BASED,
      2 (waitbit, postbit, dummy(6)) BIT(1),
      2 primitive_no  BIT(8),
      2 disposition   FIXED BIN(15);
```

A PL/MSG event signal is used both to signal the completion of an MSG pending event and to tell whether or not the event completed normally. Thus it serves both the functions of <signal> and <disp> as documented in the NSW MSG specifications (reference 1). You never need to set or clear such a block. You must provide a method of waiting on it and one of reading its value.

Notice that an ECB is different from a PL/I event variable, although an event variable does contain or point to an ECB. An event signal cannot be used directly in a PL/I WAIT statement, but you can use any of several other methods. One alternative is the MSGWAIT subroutine, explained in the section entitled SUPPLEMENTARY SUBROUTINES. You may also test for the completion of a pending event with a statement like:

```
IF ADDR(event_signal)->postbit
   THEN /* event-complete processing */
```

In any case, do not assume that a pending event is complete, and do not alter its supplied parameters or use its returned values until you have detected PL/MSG's signal.

The "disposition" portion of the event signal is set simultaneously with POSTBIT. You can test it with a statement of the form:

```
IF ADDR(event_signal)->disposition ¬= SUCCESSFUL
   THEN /* abnormal completion processing */
```

The values of "disposition" are defined uniformly across all PL/MSG calls. The PL/I programmer should not concern himself with the numeric values of this datum, but should associate names with them through %INCLUDE segment MSGDISP. See the section entitled CANNED PL/I DECLARATIONS for a listing of that segment, and the section entitled EXCEPTIONAL CONDITIONS for the meanings of the currently defined names.

The portion of the event signal called "primitive_no" in the above declaration is set by PL/MSG to indicate the class of pending event which is being signalled. Most PL/MSG callers will not need to examine this; however, it will be useful for debugging. Its values are actually fixed binary numbers. The PL/I programmer should not concern himself with the numeric values of this datum, but should associate names with them through %INCLUDE segment MSGPNO. Currently defined names listed in the section entitled CANNED PL/I DECLARATIONS.

3.5.4. MESSAGE AREAS

A PL/MSG message area is a datum of the form:

```
DCL message_area CHAR(nn) VAR;
```

where "nn" is chosen to be compatible with the calling program's operation. However, it is useful to redeclare it in the following form:

```
DCL message_length BASED FIXED BIN(15) UNALIGNED;
```

When passing a message to PL/MSG, the normal process of assigning a value to the string will set its length field properly. When requesting a message from PL/MSG, the calling program must specifically set the length field to the maximum length that it is willing to accept -- normally "nn". The most efficient way to do this is:

```
ADDR(message_area)->message_length = nn;
```

3.5.5. TIMEOUT INTERVALS

A PL/MSG timeout interval is a datum of the form:

```
DCL timeout FIXED BIN(31);
```

It is set by the caller of a PL/MSG entry that creates an MSG pending event, and specifies a time interval, in hundredths of real seconds, after which the request will be aborted and so signalled. A value of zero represents an infinite interval.

3.5.6. SPECIAL HANDLING CODES

A PL/MSG special handling code is a structure of the form:

```
DCL 1 special_handling ALIGNED,
      2 (sequenced, stream_marker) BIT (1) UNALIGNED;
```

If a bit is set, it requests or reports the use of the correspondingly named MSG special handling feature for an associated message.

3.5.7. ALARM CODES

A PL/MSG alarm code is a datum of the form:

```
DCL alarm_code FIXED BIN(15);
```

Alarm codes take on non-negative integer values.

3.5.8. WAIT ENABLES

A PL/MSG wait enable is a datum of the form:

```
DCL wait_enable BIT (1) ALIGNED;
```

A value of '0'B indicates that a sendgenericmessage is not to remain pending if no receivegenericmessage is already pending for it.

3.5.9. CONNECTION REQUESTS

A PL/MSG connection request is a structure of the form:

```
DCL 1 connection_request,
      2 type          CHAR (4),
      2 byte_size     FIXED BIN(15),
      2 connection_id FIXED BIN(15),
      2 queue_depth   FIXED BIN(15);
```

where "type" is one of the strings:

* 'STEL' for a server telnet connection;

* 'UTEL' for a user telnet connection;

* 'FULL' for a binary full duplex connection;

* 'SEND' for the sending end of a binary simplex connection;

* 'RECV' for the receiving end of a binary simplex connection.

* 'TCAM' for the special UCLA variant on STEL.

"Byte_size" may be between 1 and 255, but will probably usually be 8 or 32. Byte sizes of less than 8 bits present special problems to the programmer, and are thus discouraged. This value is ignored for telnet connections, which are always of width 8.

"Connection_id" is a caller-supplied connection number known to a remote process. The remote process must specify the same value in order to connect the two ends of the connection properly.

"Queue_depth" states the maximum number of get and put events that can be pending on the connection at any given time. This must be a small positive integer.

3.5.10. CONNECTION HANDLES

A PL/MSG connection handle is a datum of the form:

```
DCL connection_handle POINTER;
```

The value of this datum is returned from openconn, and is used in subsequent connection operations to identify the connection.

3.5.11. ALARM ARMS

A PL/MSG alarm arm is a datum of the form:

```
DCL alarm_arm BIT (1) ALIGNED;
```

A value of '1'B indicates that alarms can be accepted. '0'B indicates that they should be rejected.

3.6. THE MSGBUG CONTROL INTERFACE

PL/MSG interfaces with the using MSG process via the subroutine calls described earlier in this document, and with the MSG section of the UCLA NCP via a protocol that is strictly internal. For MSG process control and debugging, a third interface, MSGBUG, has been added.

MSGBUG is a human interface directly to the session-controlling TSO terminal. It is present in all PL/I programs that use PL/MSG, but its capabilities are limited by the environment in which the program executes.

MSGBUG can accept orders from the command that initiates execution of the program or from the session-controlling terminal. While the former path is available to all running programs, the latter is available only to TSO programs, and of real use only to those whose session-controlling "terminal" is not MSG's process-creation mechanism. The logging features of MSGBUG permit output to be directed to files and/or through a non-associated real terminal, so logging can be useful in any environment.

3.6.1. MSGBUG ORDERS

The set of MSGBUG orders is intended to be easily expandable. The present set is minimal, and is known to be insufficient, particularly in the areas of direct-connection support and actual status modification. MSGBUG will usually operate under the TSO TEST processor, so TEST functions have not been duplicated except where specifically advantageous.

There are two basic kinds of orders: those which are acceptable only at program initialization, and those which can be entered at any time during program operation. In the lists below, the order of presentation is important -- see the section entitled "ORDER PRIORITY".

3.6.1.1. INITIALIZATION ORDERS

These orders can be entered on an EXEC card or a TSO command that initiates execution of a load module containing MSGBUG. If they are entered through an attention interruption, they will be rejected.

* SHADOW/NOSHADOW enables or disables the materialization of a "shadow process" for every MSG process materialized. A shadow process is one whose generic name is the userid under which the program is operating. The only activity in a shadow process is the monitoring of a single "receivegenericmessage" primitive. If this primitive is ever completed, that is, if anyone ever sends a generic message to the shadow process, it will immediately execute the MSGBUG STOP order. The purpose of this

is to provide a primitive "cancel with dump" capability.

* SYSTEM(suffix) alters the 1-character suffix that identifies which MSG central the process will communicate with. When an installation runs more than one MSG, they are distinguished by unique 1-character suffixes.

* SUBPOOL(number) changes the place where PL/MSG will allocate its internal storage.

* INSTANCE(number) causes the process to materialize with a specific instance number, instead of a nuewly-assigned one. This order is for supporting "process introduction", and is not functional in the present implementation. It is accepted, however.

* STAX/NOSTAX enables or disables the capability of the terminal user to escape from the MSG program to a higher level program, such as TEST, the TSO TMP, or MSGBUG itself, through the attention interruption. NOSTAX effectively disables MSGBUG, and also contains an NSW tool user within the NSW system.

* EPTRACE/NOEPTRAC enables or disables the MSGBUG entry-point trace. If this trace is enabled, PL/MSG intercepts control when ANY PL/I routine is called within the task, and writes the entry-point identifier via TPUT.

3.6.1.2. DYNAMIC ORDERS

These orders can be entered during program execution, through an attention interruption. They can also be entered at initialization; however, some of them make little sense at that point.

* HELP prints a list of valid order names (but does not explain them).

* POST(address) issues an OS POST macro against the address given. The post code will be the one named TIMEOUT_EXCEEDED.

* FORCE(number) forces completion of a pending event, with code TIMEOUT_EXCEEDED. The "number" is the one associated with the event by the trace function, which can be learned through a RECENT order (see TRACE).

* TERSE/VERBOSE sets the mode of output for all subsequent RECENT or STATUS orders, or from the logging function. If TERSE is specified, the output will use highly-abbreviated annotations that are geared to the seasoned MSGBUG user. Use of TERSE can halve the volume of MSGBUG terminal output. If VERBOSE is specified, the outputs will use unabbreviated annotations that are largely self-explanatory.

* TCAM/NOTCAM determines whether the special TCAM form of the TELNET direct connection will be supported for this process. A request for a TCAM connection when NOTCAM is in force will be interpreted as a request for an STEL connection.

* LOG/NOLOG turns on and off the logging function. Events occurring when LOG is in effect will be written to whatever destinations are specified by the JONLINE, JUSER, and JFILE orders. When NOLOG is in effect no logging data is written to any destination.

* TRACE[(traceno)]/NOTRACE control the MSG tracing function. TRACE allocates a circular trace table "traceno" MSG events in circumference. If "traceno" is omitted, the last-specified value is used, and if none has ever been specified, a default of 20 is used. If "traceno" is specified, it is remembered for possible future reuse.

  NOTRACE turns off the MSG tracing function and releases any allocated trace table storage. It does not alter the remembered value of "traceno".

  If TRACE is issued while the tracing function is active, NOTRACE is simulated first. This can be used to change the size of the trace table, but only at the expense of losing its current contents. TRACE is required to use RECENT or FORCE.

* STATUS prints messages defining the current status of the MSG process, including details about each pending event.

* DCTRACE[(dctraceno)]/NODCTRACE controls the direct-connection trace function. DCTRACE turns it on, and sets the size of each connection's circular trace table to "dctraceno". If "dctraceno" is omitted, the last-specified value is used, and if none has ever been specified, a default of 20 is used. If "dctraceno" is specified, it is remembered for possible future reuse.

  NODCTRACE turns off the direct-connection trace function for direct connections opened subsequently. It does not alter the remembered value of "dctraceno".

  The direct-connection trace function is fixed for each direct connection at the time the corresponding openconn primitive is issued, and remains fixed for that connection until it is closed. Neither order affects tracing operations for currently open direct connections. If DCTRACE is issued while the direct-connection tracing function is already active, only the value of "dctraceno" will be affected. If this is omitted, the order will be a no-operation.

* RECENT[(n)] prints the newest "n" entries in the MSG event trace table. If "(n)" is omitted, the entire table is printed. The order of printing is always newest to oldest. RECENT requires that the tracing function be active.

* SNAP causes the MVT task SNAP service to be invoked against the MSG-using task. Upon successful completion of this order, a data set will have been created, filled with snap data, and enqueued for immediate printing. The routing of the printed output is determined by the path established for the TSO userid under which this session is logged on.

* JONLINE/NOJONLIN enables or disables the "online" destination for log data. If this destination is enabled, all log data lines will be written to the controlling terminal via TPUT.

* JFILE/NOJFILE enables or disables the "file" destination for log data. If this destination is enabled, all log data lines will be written to a file with DDNAME MSGJOUR.

* JUSER(logid)/NOJUSER enable and disable the "remote terminal" destination for log data. If this destination is enabled, all log data lines will be written to the terminal signed on as "logid", if it is, in fact, logged on. This option enables real-time monitoring of NSW activity within processes operating without a real TSO terminal.

* JTYPES[(mask)] specifies a filter for the various types of log output. "Mask" is a 1- to 16-bit string, where the bits are numbered 0 - 15, and correspond to the message types as defined by %INCLUDE segment MSGJOUR. For each bit, a '1' enables logging of the corresponding message type, and a '0' disables it.

* CONTINUE/STOP control the resumption of operation after MSGBUG has executed a string of orders. If CONTINUE is specified, control returns to the executing process. If STOP is specified, MSGBUG issues an "ABEND 999,DUMP" macro. If neither is specified, MSBBUG prompts for more orders.

3.6.2. ENTERING MSGBUG ORDERS

MSGBUG accepts orders from, and prints replies to, the controlling TSO terminal. Orders are entered in groups of individual order names separated by blanks or commas, and terminated by a semicolon, carriage return, or newline. To be precise, they follow the syntax of keyword operands of a TSO command, as defined in reference 2. Orders can be entered in two situations:

1) As operands of the TSO command that invokes the program that will materialize the process. Status switching orders are particularly valuable here, as this is the place that MSGBUG functions are usually turned on in the first place (to keep production processes efficient, all such functions are turned off by default). Information requests are legal, but they have no useful information to report at this point.

   Assuming any appropriate task libraries are already established, the format of a TSO command to invoke a PL/I program using PL/MSG is:

   <program> <blank> <orders> ; <options> / <parameters>

   where:

   "program" is the TSO command name, which is also the name of the program to be executed, as it has been stored in load module form.

   "orders" is a group of MSGBUG orders.

   "options" is PL/I running system execution-time options.

   "parameters" is whatever the MAIN PL/I procedure expects as its parameter string.

   For example:

   FILEPKG LOG,TRACE(50);ISA(4K)/GENNAME='FP'

2) As a stand-alone string, in response to the MSGBUG mode message (this path can be disabled by the NOSTAX order entered through the other path). You attain the MSGBUG control level by signalling the TSO attention function, however that is defined for your particular terminal. As TSO session control levels go, the current implementation of MSGMODE is primitive. The following notes may be helpful:

* "Attention" does not function as a line-delete only character. If you signal attention at any time on the MSGBUG control level, you will be transferred to the next higher control level (usually TEST or READY), and anything else typed on the current line will be discarded.

* On return from a higher control level to MSGBUG, any outstanding MSGBUG mode message is not repeated.

* If MSGBUG was attained while the using program was waiting for terminal input, returning to the original control level may satisfy that wait with a null input string.

* Remember that any TSO control level escape via "attention" always purges any enqueued terminal output buffers.

3.6.3. MSGBUG DEFAULTS

Until an explicit MSGBUG order is issued, the default state will be:

NOSHADOW,STAX,NOEPTRACE,VERBOSE,NOLOG,NOTRACE,NODCTRACE

If no explicit value is ever given for the "traceno" or the "dctraceno" operands of TRACE or DCTRACE, a default of 20 will be used.

3.6.4. ORDER PRIORITY

Any number of MSGBUG orders can be entered on a single command line. The order in which they are processed follows two rules:

* For confliciting parameters (as LOG/NOLOG in the list above), only the last entered is recognized.

* For non-conflicting parameters, regardless of the order of entry, the processing order is that in which the orders are introduced in the previous section.

3.6.5. MONITORING MSGBUG REMOTE LOGGING

To monitor the log output from an MSG process that is operating with a "JUSER(xxx)" order in effect, simply log onto TSO as userid "xxx". The log stream will be typed on your terminal. It is very important, when you are logged on in this fashion, not to delay the log output stream, since this can stop execution of the process doing the logging. If your terminal has a "locking keyboard", either in fact or virtually, the keyboard must be locked before output can be written to the terminal. If necessary, execute a command which merely goes into an endless wait pending an attention interrupt.

3.7. IMPLEMENTATION PARTICULARS

3.7.1. MATERIALIZING MULTIPLE PROCESSES

In order to materialize more than one process, you must declare and manage a STATIC EXTERNAL datum with the name PROCESS and the format of a "process handle". This datum is a handle to PL/MSG's tables that drive a process. It is set by MSGMP with the same value that that call returns in its "process handle" parameter. So long as this value is unchanged, subsequent PL/MSG calls will be on behalf of that process.

In order to function as several processes concurrently, a program must maintain its own set of process-control blocks, where each contains the associated process handle. This value should be restored to PROCESS whenever the program changes its process identity.

Each process materialized will remain known to MSG until a stopme primitive is executed with the corresponding value in PROCESS, or until the task that was in control at process materialization terminates.

Notice that, due to PL/MSG's use of static storage to communicate the process handle, a load module containing PL/MSG cannot be reentrant. This means that such a program MUST do its own "process management". You cannot rely on external multi-tasking support, even if your program is otherwise reentrant.

3.7.2. THE PENDING EVENT SET

The UCLA MSG implementation is not very strict about the MSG pending event set. There is a limit of one each of the receive-class events: receivealarm, receivegenericmessage, and receivespecific message. Beyond that, there is a limit of 25 on the total number of concurrent PL/MSG internal activities. It is not possible to define these internal activities in terms of PL/MSG concepts; however, the following approximation will always keep you safe:

* Count one internal activity for each MSG pending event. This does not include MSGGET or MSGPUT.

* Count one internal activity for each opened direct connection whose close ECB has not been posted.

* Count one internal activity for resync. The only way of knowing when such an activity is completed is when a message that would have been blocked has completed.

* Count one internal activity for rescind. This activity is completed when the pending event which is the object of the rescind primitive is posted.

3.7.3. BUFFER LENGTH BOUNDS

All PL/MSG buffers are constrained by their PL/I-compatible representation to have lengths in the range 0-32767. PL/MSG further requires that such lengths be non-zero. In the case of direct connection buffers, any lengths in the range 1-32767 can be handled fairly efficiently. In the case of buffers for MSG messages, the upper bound on tolerable buffer length depends on network activity, and cannot be stated precisely. In general, message buffers of more than 1000 bytes are considered unreasonable.

3.7.4. CODE TRANSLATION

By ARPANET convention, character transmission uses the ASCII-68 character set. Since PL/MSG is not sensitive to the forms of messages that it handles, character parts will be in ASCII in the PL/MSG caller's storage. It is up to the caller to provide appropriate translations in both directions (see reference 3).

The character portions of process names that are handled separately by PL/MSG, that is, that are not embedded in MSG messages, are always expressed in EBCDIC in the caller's storage.

Data transmitted over a direct connection is encoded by mutual agreement between the two owners of the connection, so it is your responsibility to determine this and act accordingly. Telnet-type connections are exceptions. Unless you have arranged other options with the UCLA telnet protocol handlers, translation is handled by those routines, and telnet data in your storage is always in EBCDIC. The mechanism for setting options in the telnet protocol is independent of MSG, and is documented elsewhere (see reference 4).

3.7.5. EXCEPTIONAL CONDITIONS

The PL/MSG exceptional conditions of interest correspond to the names of the values returned in event signals. These values are defined by %INCLUDE segment MSGDISP (see the section entitled CANNED PL/I DECLARATIONS). The meanings of these names are listed here.

* SUCCESSFUL -- the event completed normally.

* ALREADY_PENDING -- the set of pending events already includes as many events of this class as is permitted. See the section entitled THE PENDING EVENT SET.

* BAD_LENGTH_BUFFER -- A message area which is to receive a MSG message has a maximum length field which is either zero or greater than the implementation permits. See the section entitled BUFFER LENGTH BOUNDS.

* BAD_BYTE_SIZE -- a connection byte size is not in the range 1-255.

* BAD_CONN_TYPE -- a connection type code is unknown.

* MESSAGE_TOO_LONG -- An incoming message was longer than the maximum length that the caller was willing to accept. The message has been lost.

* UNKNOWN_OPTION_BITS -- This code cannot occur with the current version of PL/MSG.

* RESCINDED -- the pending event was aborted in response to a rescind primitive.

* NAME_TABLE_LOAD_ERROR -- an error has occurred within the internal management of the MSG task. If this occurs, MSG needs maintenance.

* BAD_HOST -- the host number field of a process name contains a number that is not assigned to an NSW host computer.

* BAD_INCARNATION -- the MSG of the host being addressed has been reincarnated, and the requested process no longer exists under the given name.

* BAD_INSTANCE -- the addressed instance of the process no longer exists.

* BAD_GENERIC_LENGTH -- the length of a generic name string is not in the range 1 - 127.

* BAD_GENERIC_NAME -- the generic name string is not recognized by the addressed host.

* MUST_RESYNC -- transmission of this type of message to this process is inhibited due to a failure to deliver a sequenced or stream-marked message. The operation might be successful if retried after an appropriate resynch primitive is issued.

* UNKNOWN_PRIMITIVE -- there has been a communication breakdown between PL/MSG and the network MSG task. This probably means that you are using an out-of-date PL/MSG package, or that your program has destroyed PL/MSG storage.

* UNKNOWN_GENERIC_CODE -- The process class of a sendgenericmessage is unknown at the requested host.

* ALARM_NOT_ACCEPTED -- the process to which an alarm was sent is not armed for alarms.

* ALARM_QUEUED -- the process to which an alarm was sent does not have an enabled receivalarm primitive pending, but is armed for alarms. The alarm was queued.

* UNKNOWN_DESTINATION -- the host number field of a process name contains a number that is not assigned to an NSW host computer.

* NO_RECEIVE_PENDING -- a sendgenericmessage cannot be associated with a receivegenericmessage without being queued. Since the caller has requested that this not be done, the message has been rejected.

* TIMEOUT_EXCEEDED -- a primitive failed to complete within the time specified as its timeout value. It has been aborted.

* DESTINATION_QUEUE_FULL -- the intended recipient of an outgoing MSG message is backlogged and is rejecting messages.

* MY_PROCESS_NAME_BAD -- an attempt has been made to materialize a local process with an invalid generic name.

* NO_RESOURCES_TO_START -- a process must be started to satisfy a sendgenericmessage, but the host is saturated in one way or another, and new processes cannot be started just now.

* UNKNOWN_ERROR -- what you were asking for didn't work, but no one is willing to venture just why.

* DESTINATION_HOST_DIED -- the addressed process resides on a host which is reported by the network to be down.

* DUPLICATE_CONNECTION -- an openconn has been requested for a remote process and connection id for which a connection already exists. Closeconn has not been completed for the old connection.

* NO_RESOURCES_FOR_CONN -- not all the mechanisms needed to satisfy an openconn are presently available.

* PROCESS_REQUESTED_CLOSE -- a direct connection has been closed normally by one or the other of the using processes.

* CONNECTION_ABORTED -- a direct connection has been closed abnormally due to a failure in the network communication mechanisms that support it.

* NO_SUCH_CONNECTION -- an operation has been requested specifying a direct connection which does not exist.

* NO_SUCH_PROCESS -- an operation has been requested specifying a process name that does not stand for a known process.

* BAD_CONNECTION_ID -- a direct connection cannot be opened because the remote process is requesting a different connection identifier.

* TYPE_MISMATCH -- a direct connection cannot be opened because the remote process is requesting an incompatible connection type code.

* BYTE_MISMATCH -- a direct connection cannot be opened because the remote process is requesting a different connection byte size.

* CANNOT_OPEN_CONNECTION a direct connection cannot be opened because of any other reason.

* TRANSMISSION_ERROR -- a connection has been closed in such a way that PL/MSG cannot guarantee that all transmitted data was delivered first.

3.7.6. INSERTING PL/MSG INTO A MODULE

PL/MSG will be included in the load module of any program which references its entry points, provided the appropriate subroutine libraries are available to the Linkage Editor. However, the module entry point must be altered to support MSGBUG, and this is not automatic. To ensure a correct entry point, add these statements to the Linkage Editor input (this example is only for a PL/I program to be executed as a TSO command):

```
INCLUDE SYSLIB(MSGSTART)
ENTRY MSGSTART
```

In addition to the modules bound at linkage-edit time, the module with aliases MSGSTAX, MSGCMD, and MSGEDIT must be available for dynamic loading during program execution. This means that the library containing it must be a part of the task library structure, as for example via a STEPLIB card in your LOGON procedure.

3.8. CANNED PL/I DECLARATIONS

For the convenience of the PL/I programmer, certain declarations will be stored in a public library. These can be invoked through the PL/I %INCLUDE statement. Because unneeded entry point declarations can result in the inclusion of extra subroutines in the final load module, each PL/MSG entry has been stored under its own name. You should select just those that you intend to use; for example:

```
%INCLUDE MSGRGM, MSGSSM, MSGDISP;
```

The available %INCLUDE names are listed below;

%INCLUDE segment MSGAA contains:

```
DCL MSGAA ENTRY (BIT (1) ALIGNED)
      OPTIONS (ASSEMBLER, INTER);
```

%INCLUDE segment MSGCC contains:

```
DCL MSGCC ENTRY (POINTER);
```

%INCLUDE segment MSGDISP contains:

```
DCL 1 MSGDISPOSITION STATIC,
      2 ERROR_TEXT (0:38) CHAR (24) VAR INIT
                    ('SUCCESSFUL',
                     'ALREADY PENDING',
                     'BAD LENGTH BUFFER',
                     'BAD BYTE SIZE',
                     'BAD CONN TYPE',
                     'MESSAGE TOO LONG',
                     'UNKNOWN OPTION BITS',
                     'RESCINDED',
                     'NAME TABLE LOAD ERROR',
                     'BAD HOST',
                     'BAD INCARNATION',
                     'BAD INSTANCE',
                     'BAD GENERIC LENGTH',
                     'BAD GENERIC NAME',
                     'MUST RESYNC',
                     'UNKNOWN PRIMITIVE',
                     'UNKNOWN GENERIC CODE',
                     'ALARM NOT ACCEPTED',
                     'ALARM QUEUED',
                     'UNKNOWN DESTINATION',
                     'TOO MANY PROCESSES',
                     'NO RECEIVE PENDING',
                     'TIMEOUT EXCEEDED',
                     'DESTINATION QUEUE FULL'
                     'MY PROCESS NAME BAD',
                     'NO RESOURCES TO START',
                     'UNKNOWN ERROR',
                     'DESTINATION HOST DIED',
                     'DUPLICATE CONNECTION',
                     'NO RESOURCES FOR CONN',
                     'PROCESS REQUESTED CLOSE
                     'CONNECTION ABORTED',
                     'NO SUCH CONNECTION',
                     'NO SUCH PROCESS',
                     'BAD CONNECTION ID',
                     'TYPE MISMATCH',
                     'PYTE MISMATCH',
                     'CANNOT OPEN CONNECTION'
                     'TRANSMISSION ERROR'),
```

(continued)

```
   2 (SUCCESSFUL                INIT (0),
      ALREADY_PENDING           INIT (1),
      BAD_LENGTH_BUFFER         INIT (2),
      BAD_BYTE_SIZE             INIT (3),
      BAD_CONN_TYPE             INIT (4),
      MESSAGE_TOO_LONG          INIT (5),
      UNKNOWN_OPTION_BITS       INIT (6),
      RESCINDED                 INIT (7),
      NAME_TABLE_LOAD_ERROR     INIT (8),
      BAD_HOST                  INIT (9),
      BAD_INCARNATION           INIT (10),
      BAD_INSTANCE              INIT (11),
      BAD_GENERIC_LENGTH        INIT (12),
      BAD_GENERIC_NAME          INIT (13),
      MUST_RESYNC               INIT (14),
      UNKNOWN_PRIMITIVE         INIT (15),
      UNKNOWN_GENERIC_CODE      INIT (16),
      ALARM_NOT_ACCEPTED        INIT (17),
      ALARM_QUEUED              INIT (18),
      UNKNOWN_DESTINATION       INIT (19),
      TOO_MANY_PROCESSES        INIT (20),
      NO_RECEIVE_PENDING        INIT (21),
      TIMEOUT_EXCEEDED          INIT (22),
      DESTINATION_QUEUE_FULL    INIT (23),
      MY_PROCESS_NAME_BAD       INIT (24),
      NO_RESOURCES_TO_START     INIT (25),
      UNKNOWN_ERROR             INIT (26),
      DESTINATION_HOST_DIED     INIT (27),
      DUPLICATE_CONNECTION      INIT (28),
      NO_RESOURCES_FOR_CONN     INIT (29),
      PROCESS_REQUESTED_CLOSE   INIT (30),
      CONNECTION_ABORTED        INIT (31),
      NO_SUCH_CONNECTION        INIT (32),
      NO_SUCH_PROCESS           INIT (33),
      BAD_CONNECTION_ID         INIT (34),
      TYPE_MISMATCH             INIT (35),
      BYTE_MISMATCH             INIT (36),
      CANNOT_OPEN_CONNECTION    INIT (37),
      TRANSMISSION_ERROR        INIT (38))
                          FIXED BIN (15);
```

%INCLUDE segment MSGEOD contains:

```
    DCL MSGEOD ENTRY (POINTER);
```

%INCLUDE segment MSGGET contains:

```
DCL MSGGET ENTRY (POINTER,
                  CHAR (*) VAR,
                  FIXED BIN (31));
```

%INCLUDE segment MSGHTPE contains:

```
DCL MSGHTYP ENTRY (FIXED BIN(15),
                   CHAR(32) VAR,
                   CHAR(32) VAR);
```

%INCLUDE segment MSGJOUR contains:

```
DECLARE MSGJOUR ENTRY (CHAR (*) VAR, FIXED BIN(15)),
        1 MSGJTYP STATIC EXTERNAL,
          2 (MJSTATISTICS       INIT (0),
             MJCOMMENT          INIT (1),
             MJALARM            INIT (2),
             MJTSO_COMMAND      INIT (3),
             MJTSO_RESPONSE     INIT (4),
             MJINCOMING_MSG     INIT (5),
             MJOUTGOING_MSG     INIT (6),
             MJINCOMING_DATA    INIT (7),
             MJINCOMING_BIN     INIT (8),
             MJOUTGOING_DATA    INIT (9),
             MJOUTGOING_BIN     INIT (10),
             MJACTION_REQD      INIT (11),
             MJECI_COMMAND      INIT (12),
             MJMSG_LOG          INIT (13),
             MJSPARE (3))       FIXED BIN(15);
```

%INCLUDE segment MSGMP contains:

```
DCL MSGMP ENTRY (1, 2 FIXED BIN (15),
                    2 FIXED BIN (15),
                    2 FIXED BIN (15),
                    2 CHAR (*) VAR,
                  POINTER,
                  FIXED BIN (31))
         OPTIONS (ASSEMBLER, INTER);
```

%INCLUDE segment MSGOC contains:

```
DCL MSGOC ENTRY (1, 2 CHAR (4),
                   2 FIXED BIN (15),
                   2 FIXED BIN (15),
                   2 FIXED BIN (15),
                 1, 2 FIXED BIN (15),
                   2 FIXED BIN (15),
                   2 FIXED BIN (15),
                   2 CHAR (*) VAR,
                 FIXED BIN (31),
                 POINTER,
                 FIXED BIN (31),
                 FIXED BIN (31));
```

%INCLUDE segment MSGPNO contains:

```
DCL 1 MSGPRIMITIVES STATIC,
   2  NAME(0:13)  CHAR(3) INIT(
        'NON',
        'RGM',
        'SGM',
        'RSM',
        'SSM',
        'RA ',
        'SA ',
        'OC ',
        'STP',  /* TERMINATION SIGNAL (STOP) */
        'AA ',
        'SYN',   /* RESYNC */
        'RSD',   /* RESCIND*/
        'GET',
        'PUT' ),
   2 (NON_MSG_EVENT           INIT (0),
      RECEIVEGENERICMESSAGE   INIT (1),
      SENDGENERICMESSAGE      INIT (2),
      RECEIVESPECIFICMESSAGE  INIT (3),
      SENDSPECIFICMESSAGE     INIT (4),
      RECEIVEALARM            INIT (5),
      SENDALARM               INIT (6),
      OPENCONN                INIT (7),
      TERMINATIONSIGNAL       INIT (8),
      ACCEPTALARM             INIT (9),
      RESYNCH                 INIT(10),
      RESCIND                 INIT(11),
      MSGGET                  INIT (12),
      MSGPUT                  INIT (13))
                         FIXED BIN (15);
```

%INCLUDE segment MSGPUT contains:

```
DCL MSGPUT ENTRY (POINTER,
                  CHAR (*) VAR,
                  FIXED BIN (31));
```

%INCLUDE segment MSGRA contains:

```
DCL MSGRA ENTRY (FIXED BIN (15),
                 1, 2 FIXED BIN (15),
                    2 FIXED BIN (15),
                    2 FIXED BIN (15),
                    2 CHAR (*) VAR,
                 FIXED BIN (31),
                 FIXED BIN (31))
        OPTIONS (ASSEMBLER, INTER);
```

%INCLUDE segment MSGRGM contains:

```
DCL MSGRGM ENTRY (CHAR(*) VAR,
                  1, 2 FIXED BIN (15),
                     2 FIXED BIN (15),
                     2 FIXED BIN (15),
                     2 CHAR (*) VAR,
                  FIXED BIN (31),
                  FIXED BIN (31))
          OPTIONS (ASSEMBLER, INTER);
```

%INCLUDE segment MSGRSM contains:

```
DCL MSGRSM ENTRY (CHAR(*) VAR,
                  1, 2 FIXED BIN (15),
                     2 FIXED BIN (15),
                     2 FIXED BIN (15),
                     2 CHAR (*) VAR,
                  FIXED BIN (31),
                  FIXED BIN (31),
                  1,
                    2 BIT (1) UNALIGNED,
                     2 BIT (1) UNALIGNED)
          OPTIONS (ASSEMBLER, INTER);
```

%INCLUDE segment MSGRSNC contains:

```
DCL MSGRSNC ENTRY(1, 2 FIXED BIN (15),
                    2 FIXED BIN (15),
                    2 FIXED BIN (15),
                    2 CHAR (*) VAR)
          OPTIONS (ASSEMBLER, INTER);
```

%INCLUDE segment MSGRSND contains:

```
DCL MSGRSND ENTRY (FIXED BIN (31))
         OPTIONS (ASSEMBLER, INTER);
```

%INCLUDE segment MSGSA contains:

```
DCL MSGSA ENTRY (FIXED BIN (15),
                 1, 2 FIXED BIN (15),
                    2 FIXED BIN (15),
                    2 FIXED BIN (15),
                    2 CHAR (*) VAR,
                 FIXED BIN (31),
                 FIXED BIN (31))
        OPTIONS (ASSEMBLER, INTER);
```

%INCLUDE segment MSGSETP contains:

```
DCL MSGSETP ENTRY (POINTER, BIT(*) VAR)
       OPTIONS (ASSEMBLER, INTER);
```

%INCLUDE segment MSGSGM contains:

```
DCL MSGSGM ENTRY (CHAR(*) VAR,
                  1, 2 FIXED BIN (15),
                     2 FIXED BIN (15),
                     2 FIXED BIN (15),
                     2 CHAR (*) VAR,
                  FIXED BIN (31),
                  FIXED BIN (31),
                  BIT (1) ALIGNED)
         OPTIONS (ASSEMBLER, INTER);
```

%INCLUDE segment MSGSSM contains:

```
DCL MSGSSM ENTRY (CHAR(*) VAR,
                  1, 2 FIXED BIN (15),
                     2 FIXED BIN (15),
                     2 FIXED BIN (15),
                     2 CHAR (*) VAR,
                  FIXED BIN (31),
                  FIXED BIN (31),
                  1,
                    2 BIT (1) UNALIGNED,
                    2 BIT (1) UNALIGNED)
          OPTIONS (ASSEMBLER, INTER);
```

%INCLUDE segment MSGSTOP contains:

```
DCL MSGSTOP ENTRY
       OPTIONS (ASSEMBLER, INTER);
```

%INCLUDE segment MSGWAIT contains:

```
DCL MSGWAIT ENTRY
       OPTIONS (ASSEMBLER, INTER);
```

REFERENCES

[1] NSW Protocl Committee, "MSG: The Interprocess Communication Facility for the National Software Works". Massachusetts Computer Associates Inc. document no. CADD-7601-2611, January 23, 1976.

[2] IBM Corporation, "IBM System/360 Operating System: Time Sharing Option Command Language Reference". IBM document order number GC28-6732.

[3] Braden and Ludlam, "PL/B8 -- A PL/I Interface Package for NSWB8". UCLA document UCNSW-403, November 15, 1980.

[4] Feinler and Postel (eds.), "Arpanet Protocol Handbook". Network Information Center document NIC 7104, January 1978.

[5] Rivas, Ludlam, and Braden, "An Implementation of the MSG Interprocess Communication Protocol". UCLA document TR-12, May, 1977.

[6] Worth, "ARPA TELNET". UCLA document S-202, April 18, 1977.

[7] Tolmei, "Socket 1". UCLA document S-204, May 3, 1977.

PART IV

PL/B8 -- A PL/I INTERFACE PACKAGE FOR NSWB8

This section is separately available
as UCLA document UCNSW-403

4. PART IV: THE PL/B8 PACKAGE

4.1. INTRODUCTION

4.1.1. PCPB8

NSW processes communicate with each other by sending "messages" through the underlying interprocess communication mechanism MSG (reference 1). An MSG message is a string of 8-bit bytes. In order to achieve both conceptual clarity and implementation economy, conventions have been defined for the form and content of messages that one NSW process may send to another. These conventions are layered into two levels: data structure, and message structure.

The data structure convention of NSW is based upon the "PCPB8" protocol (reference 2, Appendix 3) developed by Postel and White at SRI-ARC. Briefly, PCPB8 defines a class of data structures consisting of nested n-ary lists of elements. Each element of a list may be an atomic data element or another list.

The abstract syntax of a PCPB8 data structure is thus equivalent to a tree, where each node is either a terminal (atomic element) or the root of an n-ary subtree. In practice, the PCPB8 data structures which actually occur can be generated by regular expressions, and contain lists nested to a depth of 2 to 5.

An atomic data element is chosen from a specific set of semantic data types -- Integer, Boolean, Character String, etc. An PCPB8 element is logically the couple: (type, value), where 'type' encodes the semantic data type of the element, and 'value' is the corresponding value. Each data type has its own encodement rules for 'value'.

For communication between NSW processes, PCPB8 data structures are encoded into 8-bit bytes in accordance with a set of rules called "NETB8".

To facilitate the programming of higher-level NSW protocols on IBM 360/370 systems, a package of PL/I-callable conversion routines called "PL/B8" has been created. The objectives for the PL/B8 routines are as follows:

* To make higher-level protocol programming independent of the specific NETB8 representation for data structures.

* To allow a PL/I program using PCPB8 to be written with the maximum clarity and mnemonic content; thus, PL/B8 allows a free choice of symbolic names for the data items.

* To make the abstract syntax of each PCPB8 data structure used by higher-level protocols readily apparent within the PL/I code.

To achieve these objectives, the PL/B8 routines support a representation for PCPB8 data structures which is natural in IBM PL/I; this representation is called "PLIB8". The basic function of the PL/B8 routines is to convert PCPB8 data structures between the NETB8 and the PLIB8 representation.

The PL/B8 routines have been designed specifically for use by PL/I programs but can also be called from assembly language. We will describe the calls and data structures here in terms of PL/I.

4.1.2. DATA TYPES IN PCPB8

A PCPB8 data structure is composed of lists of elements, where each element may be an atomic data item or may itself be a list. Each element has a 'type' attribute; list elements and atomic elements have types:

(1) EMPTY

A place-holder, without type or value.

(2) BOOLEAN

Logical value, true or false.

(3) INDEX

Small unsigned integer, < 2**16.

(4) INTEGER

Signed integer, 2's complement with 31 bits of significance.

(5) BITSTR

Bit string of n bits.

(6) ASCIST

Character string of n ASCII-68 characters.

(7) LIST

The non-atomic grouping type -- a well-formed PCPB8 data structure is a single element of type LIST.

(8) CHARST

Character string of n EBCDIC characters. In a NETB8 message sent to or received from the ARPANET, a character string element must be in ASCII representation. Within an IBM system, on the other hand, the natural character representation is EBCDIC; for example, the debugging and dump routines assume EBCDIC representation. On IBM systems, therefore, it may be desirable to keep character strings within NETB8 structures in EBCDIC except during actual ARPANET transmission. The PLB8 routines therefore allow the caller to choose when the ASCII/EBCDIC conversions are to be performed. A new 'type' code has been defined for EBCDIC character strings, to give the PLB8 routines the information they need to perform the conversion; this code is intended to be strictly internal to the IBM system, however.

4.2. OVERVIEW OF THE PL/B8 PACKAGE

To describe the operation of the PL/B8 routines, we need to define an "element sequence" or simply "sequence" as an ordered set of elements, each of which is either an atomic element or a LIST. For example, an element of type LIST in PCPB8 is logically:

('LIST', (n, S))

Thus, the 'value' for a LIST element is the pair (n, S), where n is the number of elements in the list and S= {e1, ...,en} is the sequence of n elements of the list. Note, however, that not every sequence will necessarily form an entire LIST element.

4.2.1. PL/B8 DATA FORMATS

PL/B8 uses certain standard formats for the data structures with which it deals. These conventions are as follows:

4.2.1.1. NETB8 STRINGS

The standard format for a NETB8 data structure (i.e. an PCPB8 data structure in NETB8 representation) is a varying character string. The first two bytes of such a string are its actual length. By convention, the PL/B8 routines assume that such a string actually contains a well-formed PCPB8 structure, and is therefore a LIST of elements.

4.2.1.2. NETB8 SEQUENCE

An element sequence is represented by a "sequence dope vector", which is a PL/I structure of two fullwords in the following format:

```
1  NETB8_Sequence_dopevector  aligned,
     2  String_ptr    POINTER,
     2  Seq_length    FIXED BIN(31)
```

Here 'String_ptr' is the address of the first byte of the sequence string and 'Seq_length' is the total number of bytes in the (NETB8 representation of the) sequence.

4.2.1.3. PLIB8 DATA STRUCTURE

The basic data representation in PLIB8 is based upon the following general PL/I structure form:

```
1 PLIB8_structure aligned,

    2   map( n ) CHAR(6),

    2   data_element_1 ,
    2   data_element_2 ,
    ...
    2   data_element_n ;
```

In this form, the character array 'map' contains mnemonic representations for the types of the data elements which follow; 'map(i)' is the type of 'data_element_i'. We will refer to this character array as "the map of the data structure". Recalling that an PCPB8 element is logically the couple (type,value), we see that in PLIB8 all the 'type' parts are collected together in the map, while the 'data_element' substructures contain the corresponding 'value' parts. The 'data_element' variables themselves have PL/I formats which depend upon the types, as described in a later section.

A PLIB8 data structure can be decoded as an PCPB8 structure if the value of 'n' is known. The value of 'n', also called the 'count', is always passed to a PL/B8 routine along with the PLIB8 structure. The PL/B8 conversion routines generally interpret the PLIB8 map and operate under its control.

In NSW usage of PCPB8 data structures, it is common for a particular position in the structure to be occupied by either a single element of appropriate type or by 'EMPTY'. To facilitate the parsing and creation of such structures, PLIB8 defines an 'empty value' for each of the data types.

4.2.2. PL/B8 ROUTINES

We can describe the general function of the PLB8 conversion routines without considering the various data types in detail.

Suppose we have a set of PCPB8 data elements E(1),..., E(n) in NETB8 representation, and that the corresponding elements in PLIB8 representation (defined below) are denoted by: e(1), ..., e(n), i.e. the same names in lower-case. In NETB8, a collection of elements may be a sequence: "{ E(1), ..., E(n) }" , or it may be incorporated into a LIST element: "LIST(E(1), ..., E(n))". Within the PLIB8 representation, the basic data aggregate is a structure:

"STRUC(E(1), ... E(n))", which may represent either a LIST element or a sequence depending upon its usage.

With these preliminary definitions, we can describe the functions of the four major conversion routines of PLB8.

* PUTPLB8 : STRUC(e(1),...,e(n)) -> LIST(E(1),...,E(n))

  The PUT routine converts a PLIB8 structure representing a LIST into a single well-formed PCPB8 data structure, in NETB8 representation.

* CATPLB8 : LIST(E(1),...E(k))+STRUC(e(k+1),...e(n))
  -> LIST(E(1),...,E(k),E(k+1),...E(n))

  This routine converts the elements e(k+1),...e(n) and logically catenates them onto an existing sequence within a NETB8 LIST element. Note that PUTPLB8 is a special case of CATPLB8, where the initial list is empty: LIST() .

* GETPLB8 : LIST(E(1),...E(n)) -> STRUC(e(1),...,e(n))

  This routine is the inverse of PUT, converting a complete LIST from NETB8 representation into a PLIB8 data structure.

* SCANPLB: {E(1),...E(n)} ->
  STRUC(e(1),...,e(k)),{E(k+1),...,E(n)}

  SCAN is logically the inverse of CAT, de-catenating a sequence. The SCAN routine can also be considered as a 'scanner'; thus, in the formula above the first k elements of the sequence have been scanned, and the pointer has been advanced to the first unscanned element. The remainder sequence is in fact formed by simply shifting an address pointer, not by copying the input sequence.

In general, each call to the PUT or CAT routine creates a new character string containing a PCPB8 data LIST in NETB8 representation. Often this LIST will appear as a sub-LIST in a later call to PUT or CAT, to build the enclosing LIST; at that time, the sub-LIST will be copied into the new LIST. Thus, the composition of NETB8 output strings generally takes place by copying substrings together. If a PCPB8 data structure were large, this movement of data could require considerable CPU time; in practice, however, PCPB8 structures used in NSW are small, so the CPU time for moving this data is tolerable.

As we have described above, the "normal" mode of use of PUTPLB8 is one call per (sub-)LIST. However, the caller has a great latitude in the manner in which he builds the final NETB8 list. At one extreme, he can call CATPLB8 for each and every element, building the NETB8 string one element at a time. On the other hand, multiple

levels of LIST can be built in one PUT/CAT call. This is accomplished by means of a non-primitive data element in the PLIB8 data structure which is an indirect-reference to an inferior PLIB8 data structure. Whenever one of the PL/B8 routines encounters such an indirect ("STRUC") reference within the PLIB8 map under interpretation, the routine reenters itself recursively to convert the sublist defined by the PLIB8 sub-structure.

Using these indirect references, the PL/I program could in fact build a map of the format of the entire data structure, including all its sub-lists to arbitrary depth, and call PUTPLB8 once to convert the entire nested structure. Clearly, this "one-bang" approach may be diffult unless the format of the entire structure is fixed. On the other hand, building the entire structure in one call is the most efficient in PLB8 CPU time, since each element is converted and moved into the output string only once.

The input conversion routines GET and SCAN are essentially the inverses of the output routines PUT and CAT, and the caller has corresponding choices in the decompostion of NETB8 structures. Thus, if the format of the data structure is fixed the caller can create a recursive map of the entire multi-level list structure using indirect-reference ("STRUC") elements in the PLIB8 data structures, and convert the entire structure with a single call of GETPLB8. Alternatively, the caller can specify in the PLIB8 map that LIST elements are to be copied unchanged into new NETB8 strings, which will be converted in later calls to SCAN or GET.

There are some important differences between the input and the output case. SCAN would be the exact inverse of CAT if SCAN operated from right-to-left; it is simple and efficient to peel elements off the end of a NETB8 LIST. However, PL/B8 uses the more conventional left-to-right scan, and as a result it is often more efficient to deal with sequences than with LISTs. The use of sequence dope vectors allows decomposition using pointers to the original string, without repeated copying of substrings.

The PUT and CAT routines may leave character strings in EBCDIC representation; to convert them to ASCII the program can call the routine ETOAPLB. ETOAPLB will convert a single element, or scan an entire NETB8 data structure and convert all EBCDIC character strings it finds into ASCII, changing the type codes accordingly. The GET and SCAN routines, since they are creating PLIB8 representation, convert all ASCII character strings they find into EBCDIC.

4.2.3. PLIB8 REPRESENTATION

We will now give the mnemonic strings used in the map, and the corresponding PL/I format for the data element, for each data type in the PLIB8 representation. In addition, we will give the coding of the 'empty' value for each type in PLIB8.

4.2.3.1. PRIMITIVE TYPES

PCPB8 Type	Map String	PLIB8 Data Structure
EMPTY	EMPTY	(no data_element corresponds)
BOOLEAN	BOOL	2 BOOLEAN_element, 3 (isempty, 3 istrue) BIT(1) ALIGNED, Two one-bit flags, the first on to represent 'EMPTY', the second on to represent 'true'. If 'isempty' is true, 'istrue' is undefined.
INDEX	INDEX	2 INDEX_element FIXED BIN(15) INDEX_element<0 means 'EMPTY'.
INTEGER	INTEG	2 INTEGER_element aligned, 3 integer_addr POINTER, 3 integ_value FIXED BIN(31), 'integ_addr' is a pointer to the unaligned fullword value in the NETB8 input string, or is NULL to represent 'EMPTY'. 'integ_value' is the value of the integer. If element is 'EMPTY', 'integ_value' is ignored by the PLIB8-to-NETB8 conversion and set to 0 by the NETB8-to-PLIB8 conversion routines.
BITSTR	BITS	2 BITSTR_element POINTER, This is a POINTER to a based varying bit string, or NULL if element is 'EMPTY'.
CHASTR	ASCIST	2 CHARSTR_element POINTER,

This is a POINTER to a based varying
character string containing
ASCII-encoded characters,
or NULL if the element is 'EMPTY'.

4.2.3.2. NON-PRIMITIVE TYPES

PLIB8 recognizes three atomic but non-primitive data types: an MSG process name and two variations on the character-string atomic type.

PCPB8 Type	Map String	PLIB8 Data Structure
----	PROCNM	2 Process_name, 3 (host_number, host_incarnation, process_instance) FIXED BINARY(15), 3 generic_name CHAR(127) VARYING;

Within PCPB8, a process name is represented as a bit string. Note that 'generic_name' in PLIB8 is an EBCDIC character string, and that the corresponding substring of the NETB8 bit string is an ASCII string. Appropriate character-set conversion is performed automatically by the PL/B8 routines.

An "empty" process name is represented by the structure:

(0, 0, 0, '').

CHARST	CHARS	2 CHARSTR_element POINTER,

This is just like ASCIST except that the string contains EBCDIC-encoded characters.

```
CHARUC   CHARS    2  CHARSTR_element POINTER,
```

This is just like CHARST except that the characters in the string are forced to upper case during translation.

Finally, we must consider the representation of (sub-)LISTs in PLIB8. There are actually three different representations possible for a LIST: (1) recursively using an indirect reference to an inferior PLIB8 structure, (2) as a NETB8 varying character string representing a well-formed LIST, and (3) as the sequence of element contained in the LIST. The map mnemonics and formats of these representations are as follows:

PCPB8 Type	Map String	PLIB8 Data Structure
LIST	STRUC	2 Struc_element aligned, 3 addr(PLIB8_struct) POINTER, 3 Structure_count FIXED BIN(31)

Here 'addr(PLIB8_struct)' is a PL/I POINTER containing the address of an inferior PLIB8 structure, with 'Structure_count' map and data elements.

This map entry essentially causes a recursive call upon the conversion routine involved; this recursion can continue to a depth limited only by the size of the work area supplied by the caller.

```
LIST     NETB8    2  addr(NETB8_string)  POINTER
```

This is the address of a well-formed NETB8 LIST in a VARYING character string. If the element is 'EMPTY', the POINTER is NULL.

The LIST in the NETB8 string will be copied from (or to) the conversion source string (target string, respectively).

```
LIST     SEQL     2  Sequence_LIST_element,
                    3  NETB8_sequence_dopevector,
                      4  String_ptr  POINTER,
                      4  Seq_length  FIXED BIN(31),
                    3  Count FIXED BIN(31)
```

This sequence dope vector points to the sequence of NETB8 elements which are contained in the LIST. Here 'EMPTY' is represented by the sequence dope vector (NULL,0).

Table 1: Data Types for Decode Routines

Routine:	Expects:	Containing:	As from:
GETPLB8	CHAR VAR	Just 1 list	Network messages
GETSPLB	Seq. DV	Just 1 list	SEQL, REFER (if "LIST")
SCANPLB	Seq. DV	A sequence	REFER, TAIL (updates DV)

Map:	Matches:	Produces:	Can feed product to:
NETB8	List	CHAR VAR	GETPLB8
SEQL	List	Addr,lng,knt	GETSPLB
REFER	any one	Addr,lng,type	SCANPLB,GETSPLB (if LIST)
TAIL	Sequence	Addr,lng	SCANPLB
STRUC	List	(Addr,Knt) of PLB8 struct. -- Causes recursion	

Data form:	Representation:	Contains:
String	CHAR VAR	Just 1 list
Sequence	Addr,lng	A sequence (possibly 1 list)
PLIB8	Map,descriptors	Anything

4.3. PL/I CALLS

4.3.1. INPUT CONVERSION ROUTINES GETPLB8 AND GETSPLB

The routines GET and GETS are used to convert a PCPB8 LIST from NETB8 representation into a PLIB8 structure, checking the data types against a mnemonic type map. These two routines differ only in the form of the NETB8 input string. GET expects a NETB8 LIST in standard format, i.e. as a varying character string. GETS expects an element sequence, represented by a sequence dope vector, with the sequence containing exactly one element, the LIST to be scanned. These things are summarized in Table 1.

The 'map' strings in the PLIB8 structure define the types of the successive data elements to be converted. Each input element may match the type of the corresponding map entry exactly, in which case it is simply converted. However, the input element may also be 'EMPTY', which is said to match any atomic data type "weakly". If such a weak type match occurs, an 'empty' value appropriate to that data element is stored in the PLIB8 result. A weak type match does not halt conversion. However, a special return code is given to indicate that one or more weak matches occurred, alerting the caller that he may need to handle 'empty' values within this structure.

These routines have the following calls:

```
*****************************************************************
*                                                               *
*                                                               *
*   CALL GETPLB8 ( NETB8_string,                                *
*                  addr(PLIB8_structure),                       *
*                  Count,                                       *
*                  Work_area ) ;                                *
*                                                               *
*                                                               *
*   CALL GETSPLB ( NETB8_seqdv,                                 *
*                  addr(PLIB3_structure),                       *
*                  Count,                                       *
*                  Work_area ) ;                                *
*                                                               *
*                                                               *
*****************************************************************
```

4.3.1.1. PARAMETERS TO CALL

* NETB8_string -- CHAR VARYING string

This is a varying character string containing a single complete PCPB8 LIST(...) element, in NETB8 representation. This is the format in which PLMSG messages are delivered to the user program, and the format created by a "NETB8" conversion (see below).

* NETB8_seqdv -- a structure of the form:

```
1 NETB8_Seqdv aligned,
     2   list_ptr  POINTER,
     2   length    FIXED BIN(31) ;
```

This parameter is a sequence dope vector which must define a PLB8 sequence of exactly one LIST(...) element. Such a sequence dope vector will normally define a sublist, and be created by a "REFER" map element when the enclosing list was converted.

* addr(PLIB8_structure) -- POINTER

This parameter is a POINTER to the PLIB8 structure whose map defines the types expected and whose data elements will receive the converted values.

The map PLIB8_structure.map(Count) char(6) is a list of 'Count' type mnemonics of 3-6 characters. The allowable type mnemonics are:

1) The following atomic types:

* EMPTY

* BOOL (Boolean)

* INDEX

* INTEG (Integer)

* BITS (Bit string)

* ASCIST (ASCII character string)

This element will match any character string. The result will be to give the caller a pointer to a varying character string still encoded in ASCII.

2) These composite data elements:

* PROCNM (Process Name)

  This element will match a NETB8 bit string which will be interpreted and stored as an NSW process name.

* CHARST (EBCDIC character string)

  This element will match a character string in either ASCII or EBCDIC representation, and will cause an ASCII string in 'NETB8_string' to be converted to EBCDIC, changing the type code accordingly. In any case, the result will be to give the caller a pointer to a varying EBCDIC character string.

* CHARUC (Upper-case string)

  This element will match a character string in either ASCII or EBCDIC representation, and will cause an ASCII string in 'NETB8_string' to be converted to upper-case EBCDIC, changing the type code accordingly. In any case, the result will be to give the caller a pointer to a varying EBCDIC character string.

3) Sublist conversions, matching LIST (or EMPTY) elements:

* SEQL

  This map element creates an entry of the form:

```
 2 seq_LIST_element aligned,
       3  sequence_dope_vector,
          4  string_ptr  POINTER,
          4  seq_length  FIXED BIN(31),
       3  Count         FIXED BIN(31);
```

  Here 'sequence_dope_vector' defines the sequence of elements contained within the matching LIST. This sequence can subsequently be scanned in whole or part by passing the sequence dope vector to SCANPLB (see below). If the matching input element is EMPTY, the result will be "((NULL, 0), 0)".

* NETB8

  This map element causes the character substring of the NETB8 input containing the entire LIST to be copied into a varying character string target. The result is thus a new NETB8 list in standard format which subsequently can be passed to GETPLB8 for conversion.

The corresponding PLIB8 data element must be a single POINTER variable, set to the address of the varying character string to receive the LIST. The initial length of the string is taken as its maximum length, so it will generally be necessary to preset the length of the target string. If the matching input element is of type 'EMPTY' (as opposed to an empty LIST), a character string of length zero will be created.

* STRUC

This map element points to a subsidiary PLIB8 structure, into which the matching (sub-)LIST will be converted. This essentially causes a recursive call upon GETPLB8 to perform this conversion before resuming the conversion at the current level. This sublist conversion can be performed to any depth, limited only by the size of the workarea in the original call (see below).

The matching PLIB8 data element must be initialized to the form:

```
2 struc_element aligned,
     3  addr(PLIB8_substructure) POINTER,
     3  substructure_COUNT  FIXED BIN(31),
```

These two values become the second and third parameters to the effective recursive call on GETPLB8.

If the matching data element in the input is 'EMPTY', then the high-order bit of 'struc_element.addr (PLIB8_substructure)' will be turned on; thus, the POINTER value will look like a negative integer, if a BASED FIXED BIN(31) variable is overlaid. This bit will be turned off if an actual LIST is found, and its current setting will have no effect upon subsequent uses of the data element.

4) A map element that will create a reference to any data type, for subsequent conversion:

* REFER

This element matches any type in the input. It performs no conversion, but creates a data element as the couple: (sequence-dope-vector, type-mnemonic). Here 'sequence dope vector' defines a sequence of one element, the element being matched, and 'type-mnemonic' is one of the 4-character strings:

```
EMPTY  BOOL   INDX   INTG
BITS   CHAR   LIST
```

These are abbreviated versions of the primitive-type mnemonics of PLIB8 (see section 2.3). One of these abbreviated mnemonics, if used in a PLIB8 map, will have the same effect as the corresponding full mnemonic, since the PLB8 routines use only the first 3 characters of the map mnemonics (an exception is the distinction between CHARST and CHARUC -- CHAR is interpreted as CHARST).

Specifically, the data element corresponding to a REFER map element has the form:

```
2 reference_element aligned,
      3  sequence_dope_vector,
         4  string_ptr  POINTER,
         4  length  FIXED BIN(31),
      3  type_mnemonic CHAR(4) ;
```

5) Finally, the following map element matches the sequence of elements remaining unscanned and creates a corresponding sequence dope vector:

* TAIL

This element creates a sequence of all elements remaining in a given scan. For the outermost level (i.e. 'TAIL' appears in 'PLIB8_structure') the result is all the elements left in the input, regardless of LIST nesting. If the 'TAIL' item occurs in an "inner" map reached via a 'STRUC' item, however, then the sequence will contain only the elements contained in the current LIST level, i.e. matched by this map. The resulting sequence dope vector can later be passed to SCANPLB (described below) for further scanning.

The TAIL entry must be the last entry in the map in which it appears, or an error return will occur. If there are no remaining elements to be matched, the sequence dope vector will have the form (p,0), where 'p' is a pointer to the first byte beyond the last one scanned. Note that an element of type 'EMPTY' becomes part of the tail sequence, just like any other element.

* Count -- FIXED BIN(31)

The value of this parameter, which must be in the range 0-255, is the number of elements to be converted. The caller is responsible for setting it to the number of elements in the PLIB8 structure (the PLB8 routines have no direct way to check this value). The value must also match the actual size of the LIST being converted; this is true for each recursive call of GETPLB8 as the result of a 'STRUC' conversion.

The criterion for matching lengths is different if the last element of the map under consideration is a 'TAIL' item: the LIST may then be longer than 'Count' elements, with the residual becoming the TAIL sequence.

* Work_area -- (n) FIXED BIN(31)

This parameter provides a work area, the first word of which is also used to pass a return code back to the caller. The complete rules for determining the necessary size of this area are complicated, but

```
7 + 3*d + ceil(L/2)
```

fullwords will be sufficient in all cases. Here 'd' is the depth of nesting of 'STRUC' pointers, and 'L' is the depth of nesting of LISTs in the NETB8 input.

4.3.1.2. RETURN CODES

The return code values from GET and GETS calls are given below under SCANPLB.

4.3.2. INPUT SCANNING ROUTINE SCANPLB

The SCANPLB routine is used to convert a NETB8 element sequence, i.e. a partial list, into PLIB8 representation. Thus, SCANPLB may be called repeatedly to scan successive input elements and convert them sequentially. The call of SCANPLB has the same form as a call to GETSPLB:

```
*****************************************************************
*                                                               *
*                                                               *
*   CALL SCANPLB ( NETB8_seqdv,                                 *
*                  addr(PLIB8_structure),                       *
*                  Count,                                       *
*                  Work_area ) ;                                *
*                                                               *
*                                                               *
*****************************************************************
```

4.3.2.1. PARAMETERS TO CALL

* NETB8_seqdv -- a structure of the form:

```
1 NETB8_Seqdv aligned,
      2   list_ptr  POINTER,
      2   length    FIXED BIN(31) ;
```

This is a sequence dope vector used as the scan pointer for the input NETB8 element sequence. The 'list_pointer' is the address of the first byte of the NETB8 representation of the sequence, and 'length' is the total number of bytes in the sequence. When SCANPLB is called, the sequence dope vector must have been set to the point at which the conversion is to begin; when SCANPLB exits, it will update 'NETB8_seqdv' to the first element not yet scanned.

A convenient way to create a sequence dope vector suitable for initial input to SCAN is with a 'SEQL' conversion map mnemonic. Subsequent calls to SCAN with the same NETB8_seqdv structure and Count = 1 will scan this sequence element-by-element. When the sequence is exhausted, a null sequence dope vector (P,0) will be returned, having 'length' = zero.

* addr(PLIB8_structure) -- POINTER

This parameter is a POINTER to the PLIB8 structure whose map defines the types expected and whose data elements will receive the converted values. The form and content of PLIB8_structure is the same as that described earlier for GETPLB8 and GETSPLB calls.

Note that 'STRUC' map elements cause recursive calls on GETSPLB, during which the rules of that routine hold (e.g. converting an entire LIST, with lengths matching that of the map); when control returns to the level of the initial SCAN, however, the SCAN rules hold again.

* Count -- FIXED BIN(31)

The value of this parameter, which must be in the range 0-255, is the number of elements to be converted. The user is responsible for setting it to the number of elements in the PLIB8 structure (although the PLB8 routines have no direct way to check). If the sequence is exhausted before 'Count' is satisfied, SCAN will return with a return code to indicate a short NETB8 string.

* Work_area -- (n) FIXED BIN(31)

This parameter provides a work area, the first word of which is also used to pass a return code (see below). The number of fullwords in the work area is the same as described earlier for GETPLB8.

4.3.2.2. RETURN CODES

When it returns to its caller, an input conversion routine (GETPLB8, GETSPLB, or SCANPLB) will have set the first word of the work area to a return code value. This value is 0 to indicate unqualified success, or else a non-zero code which indicates the cause of failure. In the latter case, the conversion will have been completed up to the point of failure; the exceptions to this are noted in the return code list that follows. In the case of a SCAN operation that is unsuccessful, the scan pointer (sequence dope vector) will define a sequence beginning with the bad element.

* 0 -- OK

The requested operation was successful, and all types matched exactly (no 'weak' matches or mismatches).

* 1 -- Weak Match(es)

The requested operation completed successfully, but one or more type matches were "weak".

* 2 -- Undefined NETB8 Type Code

An undefined type code was found in the NETB8 input string. The requested operation will have completed up to the bad match (possibly with one or more weak type matches).

* 3 -- Unmatched NETB8 Type Code

A type code in the input did not match, even weakly, the corresponding map mnemonic; in other words, the input data structure has a form different than expected. The input type code is, however, a valid value.

* 4 -- Count Too Small (GET, GETS only)

The number of elements in the PLIB8 map and data parts, as determined by the 'Count' parameter, is less than the actual number in the input LIST, and the map does not end with a 'TAIL' element.

* 5 -- Missing bytes

Bytes which were expected on the end of the input are missing. If the input string was not truncated on an element boundary, the last element actually converted may contain spurious information.

* 6 -- Count Too Large (GET, GETS only)

The number of elements in the input LIST is less than the number of elements in the MAP (even with the trailing TAIL element, if any, ignored).

* 7 -- Expected LIST Not Found (GET, GETS only)

The input to this call is required to be a LIST element, which was not the case.

* 8 -- NETB8 String Overflow

The varying string supplied to receive a copy of a sub-LIST under control of a "NETB8" map entry is too small to hold the entire sub-LIST.

* 9 -- Misplaced TAIL Entry

A "TAIL" entry occurred in the map before the last entry.

* 10 -- Bad Map Mnemonic

The PLIB8 map contains an unrecognizable type mnemonic.

* 11 -- Bad Parameter List

The value of Count < 0, or one of the parameters in the call is not fullword aligned. This may occur in the primary call or in a consequent recursive call.

4.3.3. OUTPUT CONVERSION ROUTINES PUTPLB8 AND CATPLB8

The routines PUTPLB8 and CATPLB8 are used to convert an PCPB8 data structure from PLIB8 representation into NETB8 representation. These routines are the logical inverse of the GETPLB8 and SCANPLB routines, respectively.

PUTPLB8 creates a complete NETB8 LIST element from the NETB8 conversions of the given PLIB8 structure.

CATPLB8 extends an existing NETPLB8 LIST, catenating onto it a new sequence of elements formed by converting the elements of the given PLIB8 structure. If the original NETB8 string is of length zero when CAT is called, then a PUT operation is performed instead (thus the PUT entry is logically unnecessary but is a slight mnemonic and programmatic convenience).

Elements being converted are not tested for the "empty" value -- such a value will be converted like any other. The only way to get an "empty" element into the NETB8 output is to use a map mnemonic of 'EMPTY'.

These routines have the following calls:

```
CALL PUTPLB8(  addr(PLIB8_structure),
               Count,
               NETB8_string,
               Work_area ) ;

CALL CATPLB8(  addr(PLIB8_structure),
               Count,
               NETB8_string,
               Work_area ) ;
```

4.3.3.1. PARAMETERS TO CALL

* addr(PLIB8_structure) -- POINTER

  This parameter is a POINTER to the PLIB8 structure defining the format and values of a sequence of PCPB8 data elements, to be converted to NETB8 representation and placed in the NETB8_string.

The map contains type mnemonics which are generally the same as for the input conversion routines described earlier. The allowable mnemonics are:

1) The atomic types: EMPTY, BOOL, INDEX, INTEG, BITS, and ASCIST.

   Note that ASCIST expects an input string already in ASCII representation. The NETB8 element will also be ASCII.

2) The composite types CHARST AND CHARUC. These mnemonics are the same as ASCIST except the input string and the NETB8 result are always in EBCDIC. No distinction is made between CHARST and CHARUC in this case.

3) The composite type PROCNM. Translation from EBCDIC to ASCII will occur.

4) The sublist types SEQL, NETB8, and STRUC.

* Count -- FIXED BIN(31)

  The value of this parameter is the number of elements to be converted. It must be set to the number of elements in the PLIB8 structure (although the PLB8 routines have no way to check this).

* NETB8_string -- CHAR VARYING string

  The converted elements are placed into this string in such a way as to create a valid NETB8 LIST. PUTPLB8 replaces any previous contents of this string with a LIST of the converted elements; CATPLB8 catenates the elements onto the LIST already contained in the string.

* Work_area -- (n) FIXED BIN(31)

  This parameter provides a work area, the first word of which is also used to provide a return code. The number of fullwords in the work area must be at least 3 + 3*d , where d is the depth of nesting of 'STRUC' pointers. Thus, if no 'STRUC' map elements occur at all, n=3 is sufficient.

4.3.3.2. RETURN CODES

When it returns to its caller, an output conversion routine (PUTPLB8 or CATPLB8) will set the first word of the work area to a return code value. Even in the case of an error return, the 'NETB8_string' will be a properly formed (although probably incomplete) LIST. The return code values are as follows:

* 0 -- OK

  The requested operation was successful.

* 7 -- Sequence Dope Vector does not Define a LIST

  Either the NETB8 input string to CATPLB8 is not a LIST, or string pointed to by a NETB8 element is not a LIST.

* 8 -- NETB8 String Overflow

  The NETB8 string being created exceeds the maximum size of the output string.

* 10 -- Bad Map Mnemonic

  The PLIB8 map contains an unrecognizable type mnemonic.

* 11 -- Bad Parameter List

  One of the first 3 parameters in the original call is not properly aligned on a fullword boundary, or Count is out of the range 0-255.

4.3.4. GET EXTENT OF NETB8 ELEMENT -- NEXTPLB

The routine NEXTPLB may be used as a utility function for writing new routines for dealing with NETB8 elements in their original form. NEXTPLB is actually used as a subroutine of GETPLB8, GETPPLB8, and SCANPLB through a private non-standard entry point named GETLENG.

Given a scan pointer defined by a sequence dope vector, NEXTPLB advances the pointer to the next element on the same level of the structure, or one level higher.

```
*************************************************************
*                                                           *
*                                                           *
*   CALL NEXTPLB( NETB8_seqdv,                              *
*                 Work_area ) ;                             *
*                                                           *
*                                                           *
*************************************************************
```

4.3.4.1. PARAMETERS TO CALL

* NETB8_seqdv -- a structure of the form:

```
1 NETB8_Seqdv aligned,
     2   string_ptr POINTER,
     2   seq_length FIXED BIN(31)
```

This is a scan-pointer for a NETB8 element sequence. The 'sublist_pointer' is the address of the first byte of the NETB8 representation of the sequence, and 'length' is the total number of bytes in the sequence. When NEXTPLB is called, the sequence dope vector must point to a NETB8 element; when NEXTPLB exits, it will update 'NETB8_seqdv' to point to the next element, if any, in (tree-walking) order. If 'NETB8_Seqdv.length' is not greater than zero in the call, NEXTPLB will exit with no action taken.

* Work_area -- (n) FIXED BIN(31)

This parameter provides a work area, the first word of which is also used to provide a return code (see below). The number of fullwords n must be at least n= 1 + ceil(L/2), where 'L' is the depth of nesting of (sub-)LISTs in the element being scanned.

4.3.4.2. RETURN CODES

When it returns to its caller, NEXTPLB will have set the first word of the work area to one of the following return codes:

* 0 -- OK

* 2 -- Undefined Type Code in Input

  An impossible type code was found in the NETB8 string.

* 5 -- Missing Bytes

  The end of the string was found before the element was ended.

4.3.5. CONVERT CHARACTER STRING REPRESENTATION

The routines ATOEPLB and ETOAPLB may be used to convert the character strings, if any, in an element from ASCII to EBCDIC or vice versa. If the element is a LIST, the entire list is scanned, including any subLISTs it may have, and all CHARST's are converted.

Normally, after building an entire NETB8 message from PLIB8 elements containing type CHARST or CHARUC, ETOAPLB should be called. It is illegal to transmit a NETB8 message that still contains EBCDIC-encoded strings.

These routines have the following calls:

```
*************************************************************
*                                                           *
*                                                           *
*   CALL ATOEPLB( NETB8_string_ptr,                         *
*                 Work_area ) ;                             *
*                                                           *
*                                                           *
*   CALL ETOAPLB( NETB8_string_ptr,                         *
*                 Work_area ) ;                             *
*                                                           *
*                                                           *
*************************************************************
```

4.3.5.1. PARAMETERS TO CALL

* NETB8_string_ptr -- POINTER

  This parameter is the address of the first byte of the NETB8 element to be converted.

* Work_area -- (n) FIXED BIN(31)

  This parameter provides a work area, the first word of which is also used for the return code. The work area contains n fullwords, where n must be at least n= 1 + ceil(L/2), and 'L' is the depth of nesting of LIST elements in the element to be converted. Thus, if 'NETB8_string_ptr' points to an atomic element or a LIST containing no sublists, L=0 and 'Work_area' can be a scalar FIXED BIN(31) variable.

4.3.5.2. RETURN CODES

The return code values are the same as for the NEXTPLB routine.

4.4. CANNED PL/I DECLARATIONS

For the convenience of the PL/I programmer using PL/B8, several useful sets of PL/I declarations have been placed in a public library. These may be inserted into the PL/I program with a %INCLUDE statement. The actual text of these text segments is on the following pages. In summary, the following segments are available:

* PLB8ENTR

  This segment contains ENTRY declarations for all PLB8 routines.

* PLB8FORM

  This segment contains useful definitions of PLIB8 data elements. In particular, it defines data elements corresponding to the type mnemonics: 'BOOL', 'INTEG', 'PROCNM', 'STRUC', 'SEQL', and 'REFER'. Also defined are initial structures containing the values TRUE, FALSE, and 'empty' for a BOOL element.

* PLB8ERRM

  This segment declares an initialized character array containing error message text appropriate for the return codes from any PLB8 routine.

%INCLUDE segment PLB8ENTR contains:

```
DCL /*********************************************************/
    /*                                                       */
    /*  DECLARE ENTRIES OF PLB8 ROUTINES                     */
    /*                                                       */
    /*********************************************************/
 PUTPLB8  ENTRY( POINTER,        /* ADDR(PL1B8_STRUC)      */
          FIXED BIN(31),         /* COUNT                  */
          CHAR(*) VARYING,       /* NETB8_STRING           */
          FIXED BIN(31)),        /* WORK AREA, RETURN CODE*/
 CATPLB8  ENTRY( POINTER,        /* ADDR(PL1B8_STRUC)      */
          FIXED BIN(31),         /* COUNT                  */
          CHAR(*) VARYING,       /* NETB8_STRING           */
          FIXED BIN(31)),        /* WORK AREA, RETURN CODE*/
 GETPLB8  ENTRY(
          CHAR(*) VARYING,       /* NETB8_STRING           */
          POINTER,               /* ADDR(PL1B8_STRUC)      */
          FIXED BIN(31),         /* COUNT                  */
          FIXED BIN(31)),        /* WORK AREA, RETURN CODE*/
 GETSPLB ENTRY( 1,
          2  POINTER,            /* ADDR OF FIRST BYTE     */
          2  FIXED BIN(31),      /* LENGTH OF SEQUENCE     */
          POINTER,               /* ADDR(PL1B8_STRUC)      */
          FIXED BIN(31),         /* COUNT                  */
          FIXED BIN(31)),        /* WORK AREA, RETURN CODE*/
 SCANPLB ENTRY( 1,
          2  POINTER,            /* ADDR OF FIRST BYTE     */
          2  FIXED BIN(31),      /* LENGTH OF SEQUENCE     */
          POINTER,               /* ADDR(PL1B8_STRUC)      */
          FIXED BIN(31),         /* COUNT                  */
          FIXED BIN(31)),        /* WORK AREA, RETURN CODE*/
 NEXTPLB ENTRY(  1,              /* SEQ DOPE VECTOR        */
          2  POINTER,            /* ADDR OF FIRST BYTE     */
          2  FIXED BIN(31),      /* LENGTH OF SEQUENCE     */
          FIXED BIN(31)),        /* WORK AREA, RETURN CODE*/
 ATOEPLB ENTRY(
          POINTER,               /* ADDR OF FIRST BYTE     */
          FIXED BIN(31)),        /* WORK AREA, RETURN CODE*/
 ETOAPLB ENTRY(
          POINTER,               /* ADDR OF FIRST BYTE     */
          FIXED BIN(31)) ;       /* WORK AREA, RETURN CODE*/
```

%INCLUDE segment PLB8FORM contains:

```
DCL /*****************************************************/
    /*                                                   */
    /*  DECLARE USEFUL FORMATS OF PL1B8 DATA ELEMENTS    */
    /*                                                   */
    /*****************************************************/
    1  PL1B8_STRUC_ELEMENT  ALIGNED,
         /* PL1B8 DATA ELEMENT FOR SUBSTRUCTURE.  */
         /* MATCHES AN NETB8 'LIST' ELEMENT, MAKES*/
         /* A RECURSIVE CALL ON CONVERSION ROUTINE*/
       2  STRUC_ADDR        POINTER,
       2  STRUC_COUNT       FIXED BIN(31),
    1  PL1B8_SEQ_DOPEVECTOR  ALIGNED,
         /* DEFINES SEQUENCE OF NETB8 ELEMENTS.   */
       2  STRING_PTR        POINTER,
       2  SEQ_LENGTH        FIXED BIN(31),
    1  PL1B8_REFER_ELEMENT ALIGNED,
         /* 'REFER' ELEMENT= SEQ OF ONE ELEMENT AND ITS TYPE */
       2  REFER_SEQ_DOPEVECTOR,
          3  STRING_PTR     POINTER,
          3  SEQ_LENGTH     FIXED BIN(31) ,
       2  REFER_TYPE        CHAR(4),
    1  PL1B8_SEQ_LIST_ELEMENT  ALIGNED,
         /* 'SEQL'  ELEMENT= (SEQ OF ELEMENTS OF LIST,COUNT) */
       2  SEQ_LIST_DOPEVECTOR,
          3   STRING_PTR     POINTER,
          3   SEQ_LENGTH     FIXED BIN(31) ,
       2  LIST_COUNT        FIXED BIN(31),
    1  PL1B8_BOOL_ELEMENT   ,
       2  ISEMPTY  BIT(1)  ALIGNED,
       2  ISTRUE   BIT(1)  ALIGNED,
    1  PL1B8_INTEG_ELEMENT ALIGNED,
       2  INTEG_ADDR        POINTER,
       2  INTEG_VALUE       FIXED BIN(31),
    1  MSG_PROCESS_NAME     ALIGNED,
       2  HOST_NUMBER       FIXED BIN(15),
       2  HOST_INCARNATION  FIXED BIN(15),
       2  HOST_INSTANCE     FIXED BIN(15),
       2  GENERIC_NAME      CHAR(127) VARYING,
    1  EMPTY_BOOL_ELEMENT   ALIGNED,
       2 ( BOOL_ISEMPTY     INIT('1'B),
           BOOL_ISTRUE      INIT('0'B) )  BIT(1),
    1  TRUE_BOOL_ELEMENT    ALIGNED,
       2 ( BOOL_ISEMPTY     INIT('0'B),
           BOOL_ISTRUE      INIT('1'B) )  BIT(1),
    1  FALSE_BOOL_ELEMENT   ALIGNED,
       2 ( BOOL_ISEMPTY     INIT('0'B),
           BOOL_ISTRUE      INIT('0'B) ) BIT(1) ;
```

%INCLUDE segment PLB8ERRM contains:

```
DCL /*********************************************************/
    /*                                                       */
    /*  DECLARE ERROR MESSAGE STRINGS FOR PL/B8 ROUTINES*/
    /*                                                       */
    /*********************************************************/
    PLB8_ERROR_MESSAGES(0:12) CHAR(25) STATIC INIT(
         'OK',
         'WEAK MATCHES',
         'NULL INPUT STRING',
         'UNDEFINED NETB8 TYPE CODE',
         'UNMATCHED NETB8 TYPE CODE',
         'COUNT TOO SMALL',
         'MISSING BYTES',
         'COUNT TOO LARGE',
         'EXPECTED LIST NOT FOUND',
         'NETB8 STRING OVERFLOW',
         'MISPLACED TAIL ENTRY',
         'BAD MAP MNEMONIC' ) ;
```

REFERENCES

[1] NSW Protocol Committee, "MSG: The Interprocess Communication Facility for the National Software Works". Massachusetts Computer Associates document CADD-7601-2611, January 23, 1976.

[2] Schantz and Millstein, "The Foreman: Providing the Program Execution Environment for the National Software Works". Massachusetts Computer Associates document CADD-7701-0111, January 1, 1977.

PART V

PL/PCP -- AN NSW PROCEDURE CALL PROTOCOL PACKAGE FOR PL/I

This section is separately available
as UCLA document UCNSW-402

5. PART V: THE PL/PCP PACKAGE

5.1. INTRODUCTION

There are three levels or aspects to the NSW Transmission protocol, or NSWTP:

* A protocol specifying the organization of messages into procedure-call-like transactions following a paradigm of the form:

  <procedure> (<arguments>) -> <results>

  We have chosen to call this aspect the "Procedure Call Protocol", or PCP.

* The message formatting and transmitting protocol called "MSG".

* An underlying data representation scheme called "NSWB8".

PCP and NSWB8 are described in Appendix 3 of reference 1, MSG is extensively documented in reference 2, and fault handling in NSWTP is outlined in reference 3.

This paper describes version 2 of PL/PCP, a subroutine package that implements the PCP aspect of NSWTP. At this writing, version 2 is only partially implemented, and thus some parts of this document do not yet apply. The reader who is concerned with programming using PL/PCP now should read the subsequent section entitled STATUS OF THE IMPLEMENTATION.

PL/PCP supports calls from programs compiled by the IBM PLI Optimizing Compiler. Separate UCLA documents describe PL/B8 (reference 4) and PL/MSG (reference 5), similar packages that implement the NSWB8 and MSG aspects. PL/PCP uses these packages, and expects its caller to use them to some extent. In this document, the reader is assumed to have some familiarity with both PL/B8 and PL/MSG. PL/PCP also uses MSG support subroutines PRINTB8 and PNAMOUT; however, knowledge of these routines is not necessary in order to use PL/PCP.

PL/PCP uses PL/MSG as its actual MSG interface. The PL/PCP caller does not use PL/MSG directly except for supplementary routines, the routines handling direct connections, and the MSGAA routine, which arms and disarms the process for MSG alarms. These routines are listed in the section entitled applicability.

5.2. APPLICABILITY

PL/PCP imposes a discipline which may be too stringent for MSG users with esoteric requirments. PL/PCP is for you if your MSG process will meet these constraints:

* The using process is prepared to deal with message flow only in PCP-specified transaction constructs.

* All messages sent or received by the process use NSWB8 as the underlying data representation.

* The process will communicate with a fixed set of remote processes at a time, such that it can refer to them by named handles.

* The process will deal with a manageable set of remote transactions at a time, such that it can refer to them by named handles.

* The process has only one identity -- that is, it will materialize as only one MSG process at a time.

* The process does not issue any PL/MSG calls other than:

  MSGAA (arm for alarms)
  MSGCC (close a direct connection)
  MSGGET (read from a direct connection)
  MSGHTYP (define a remote host)
  MSGJOUR (write to the common log files)
  MSGPUT (write to a direct connection)
  MSGOC (open a direct connection)
  MSGWAIT (wait for multiple events)

  All other PL/MSG entries are appropriated by PL/PCP through the MSGSETP facility, and will cause unpredictable behavior if issued outside PL/PCP.

5.3. ENTITIES TREATED

The PL/PCP caller must deal with the following abstract entities:

5.3.1. PROCESSES

The PL/PCP caller is a local process, and communicates with a small number of remote processes, usually on other hosts. Every PL/PCP transaction is between the local process and a particular remote process. The caller normally refers to a process by naming a "process handle" provided by PL/PCP. A process handle is a PL/I POINTER variable. See the section entitled HANDLE VARIABLES.

5.3.2. PROCEDURES

A procedure is a named service performed by a process when requested to do so by another process. Typically, a single process type may be able to execute any of several procedures. PCP is concerned with managing procedure calls from one process to another.

5.3.3. TRANSACTIONS

A transaction is a structured exchange of messages between the local process and a remote process. Usually, but not always, a transaction is a procedure call and reply pair. A "local transaction" is usually a procedure call from the local process. More precisely, it is a transaction created by a message or alarm sent from the local process, and completed by a reply from the remote process. A "remote transaction" is usually a procedure call from a remote process. More precisely, it is a transaction created by a message or alarm received from a remote process, and completed by a reply from the local process. In either situation there can be the degenerate case where no reply is required. In such a case the transaction consists of only one message, which both creates and completes the transaction.

The caller normally refers to a transaction by naming a "transaction handle" provided by PL/PCP. A transaction handle is a PL/I POINTER variable. See the section entitled HANDLE VARIABLES.

5.3.3.1. TRANSACTION TYPES

It is convenient to define a set of named transaction types, as follows:

* A CALL transaction is a procedure call which introduces two processes. The most common form of CALL uses the MSG generic addressing facility to actually create a new process to execute the proceure call. However, an existing process can receive a CALL via a specifically-addressed message if a third process has given its specific name to the caller.

* A LOG transaction is a degenerate form of CALL that does not involve a reply message. It might be used to send error messages to an error logging process.

* An ASK transaction is a procedure call addressed to an already-known process. When process introduction has been performed by a third process, a transaction may be an ASK from the point of view of the caller, but a CALL from that of the callee.

* A TELL transaction is a degenerate form of ASK that does not involve a reply message. When process introduction has been performed by a third process, a transaction may be a TELL from the point of view of the caller, but a LOG from that of the callee.

* An ALARM transaction is unique in that it does not involve a procedure call. It is a high-priority numbered signal that is addressed to an already-known process. Its reply expectations are the same as for an ASK.

5.3.3.2. TRANSACTION COMPLETION CODES

A transaction is always created by the transmission of a message or alarm. It may be completed in one of four ways:

* A REPLY completion is one in which the called procedure's normal results are delivered without error to the caller.

* An ERROR completion is one in which an error indication is delivered to the caller. It may be an error message from the called procedure, a PCP timeout, or a code from MSG.

* An AUTOMATIC completion is one in which no reply is defined. This is the normal mode for LOG and TELL transactions.

* An FREE completion is any one in which a transaction that would normally expect a reply is discarded by the replying process without any reply having been sent. This is used, for instance, when it is determined that no reply need be sent for a received ALARM.

5.3.4. MESSAGES

A message is pretty much what the name implies, except that it is constrained to be formatted according to NSWB8 and MSG conventions. All PL/PCP routines associate messages with transactions, and transactions with processes.

Messages are of several types, corresponding to the types of transactions that they create and to the modes of transaction completion that they cause. The caller sends a message when he creates or completes a transaction. He does this by calling a transaction-type-dependent PL/PCP routine, passing appropriate message parts as arguments. The parts whose contents are transaction dependent must already be encoded into a "NETB8" list as defined by the PL/B8 subroutine package.

When a message is received, a single PL/PCP analysis routine is called to determine the message type and its significance in terms of transaction processing, and to create or complete a transaction if appropriate. This routine returns decriptors of the message's various constituent parts in a transaction structure. The parts whose contents are transaction dependent will be defined by a "SEQL" dope vector as defined by the PL/B8 subroutine package (reference 4).

5.3.5. ALARMS

An MSG alarm is like a very high-priority message that consists only of a single number. PL/PCP callers may treat alarms rather like messages, except that they usually should be given priority (they are already given high priority by all the communication machinery that exists between two processes). A process must arm and disarm itself for alarms through the MSGAA routine, described in the documentation of the PL/MSG package (reference 5). No alarms can be received until MSGAA has been called at least once.

5.3.6. EVENTS

Because PL/PCP callers are transaction-oriented, the events that drive their logic should be the creation and completion of transactions, not elementary MSG events. However, PL/PCP does not wait for the completion of any events, expecting that its callers will have the need to control their own waits. Furthermore, while PL/PCP's activities simulate an independent thread of control, it is implemented as a set of closed subroutines which only receive control when CALLed from the using program. Therefore, PL/PCP must define for its caller a master Event Control Block (ECB) representing all elementary events which are themselves of no interest to the caller. Whenever this ECB is posted, the PCEXAM routine must be called to turn the event into something meaningful in terms of transaction processing. Commonly, this will be the advancement of a transaction to the next state toward its completion, in which case PCEXAM will report to its caller that it has processed a "NULL" event. If, however, the event resulted in the creation or completion of a transaction, PCEXAM will report this information.

In summary, the transaction events of interest to the caller always occur synchronously to program operation, and are not represented by ECB's, whereas the elementary events of no interest to the caller occur asynchronously and are represented by a single ECB, which must be monitored.

5.3.7. TIMEOUTS

PL/PCP timeouts are completely separate from those defined by MSG. In PL/PCP a timeout interval is applied to a local transaction, not a message, and represents the amount of time that the local process is willing to have elapse between the creation of a local transaction and its REPLY completion. If the interval expires, PL/PCP will force ERROR completion of the transaction. A timeout value of zero is defined to mean infinity, and disables timing for a transaction.

PL/PCP timeout values are stated in units of hundredths of seconds; however, this does not imply the existence of a high-precision control mechanism supporting timeouts. The resolution of this mechanism is likely to be in seconds or worse, and using programs must not be designed to require more precision.

The using routine will have one occasion to refer to a true MSG *timeout -- see the "msg timeout" parameter of the PCBEGIN routine.*

5.3.8. FAULTS

A PL/PCP fault is a set of information describing a single exceptional condition that arose during processing of a transaction. When a transaction is completed, its reply message includes a list of all faults accumulated to that point. Faults are described in detail in the section entitled FAULT HANDLING IN PL/PCP.

5.3.9. SPECIAL HANDLING CODES

MSG supports two special handling options, sequencing and stream marking. Eventually, it is expected that these may be used by PL/PCP in response to PCP constructs yet to be defined. In the meantime, they are available to the PL/PCP caller as the "special handling code", a structure of the form:

```
DCL 1 special_handling ALIGNED,
      2 (sequenced, stream_marker)
              BIT(1) UNALIGNED;
```

The PL/PCP transaction structure contains such a substructure which is used to report the presence of the options on incoming messages. Routines which send messages include the structure in their calling sequences, where setting either bit requests the corresponding MSG special handling feature. Until further specification, the PL/PCP caller should set these bits to zero.

5.3.10. HANDLE VARIABLES

A PL/PCP handle variable is a PL/I POINTER variable the value of which can be set by a PL/PCP routine. Such setting establishes a relationship between the handle variable and a process or transaction, and in subsequent communication between PL/PCP and its caller, the variable name can be used as the name of that PL/PCP entity.

5.3.10.1. TRANSACTION HANDLES

A transaction handle is defined, and its value is returned, by a PL/PCP routine which creates a transaction (PCCALL, PCLOG, PCASK, PCTELL, or PCALARM, or PCEXAM when it returns a type of "CALL", "LOG", "ASK", "TELL", or "ALARM"). It becomes undefined only when it is passed to PCFREE (note that PCFREE can be called indirectly, from PCBYE or PCEND).

Besides its use as the name of a transaction, a transaction handle points to a PL/I BASED structure which contains much information of use to the caller. This structure is defined by %INCLUDE segments PCTRANS and PCSTATE (see figures 1 and 2).

Figure 1: The PCTRANS Declaration

```
/* INTERNALS OF TRANSACTION STRUCTURE.  PROVIDE YOUR OWN      */
/* LEVEL-1 DECLARATION AND TERMINATOR.                        */
 3 TRECB FIXED BIN(31),           /* PL/PCP'S INTERNAL ECB.    */
 3 TRECBLINK POINTER,             /* POINTER TO THE MASTER ECB.*/
 3 TRUSER_PTR POINTER,            /* A POINTER VARIABLE THAT   */
            /* BELONGS TO THE CALLER.  THIS IS PROVIDED ONLY */
            /* AS A CONVENIENCE, IN CASE THE CALLER NEEDS TO */
            /* CHAIN TO TABLES OF HIS OWN.                   */
 3 TRUSER_INDEX FIXED BIN(31),    /* AN INTEGER VARIABLE THAT  */
            /* BELONGS TO THE CALLER.  THIS IS PROVIDED ONLY */
            /* AS A CONVENIENCE, IN CASE THE CALLER NEEDS TO */
            /* INDEX INTO TABLES OF HIS OWN.                 */
 3 TRENDTIME FLOAT BIN(44),       /* THE TIME WHEN THIS TRANS- */
            /* ACTION WILL TIME OUT, IN PCTOD FORM.          */
 3 TRPROCESS  POINTER,            /* ASSOCIATED PROCESS HANDLE */
 3 TRPROCEDURE POINTER,           /* POINTER TO A BASED CHAR   */
            /* VAR STRING, CONTAINING EITHER THE NAME OF THE */
            /* PROCEDURE BEING CALLED OR THE HUMAN-READABLE  */
            /* MESSAGE ASSOCIATED WITH A TRANSACTION ERROR.  */
 3 TRARGUMENTS,                   /* PROCEDURE ARGUMENTS.      */
            /* FOR A REMOTE TRANSACTION, A 'SEQL' DOPE       */
            /* VECTOR.  FOR A LOCAL TRANSACTION, A POINTER   */
            /* TO A 'NETB8' STRING, WITH THE LENGTH AND      */
            /* COUNT FIELDS UNDEFINED.                       */
   5 TRARDESCR,                   /* SUBLIST DESCRIPTOR:       */
     7 TRARADDR POINTER,          /* STRING ADDRESS            */
     7 TRARLNG FIXED BIN(31),     /* STRING LENGTH             */
   5 TRARCOUNT FIXED BIN(31),     /* LIST COUNT                */
 3 TRBUFANCHOR POINTER,           /* ANCHOR FOR INTERNAL TEXT  */
            /* BUFFERS MANAGED BY PL/NTP ROUTINES.           */
 3 TREXCEPTION_CLASS  FIXED BIN(15), /* ERROR TYPE CODE --    */
            /* SET TO NON-ZERO ONLY WHEN A TRANSACTION ERROR */
            /* OCCURS.  THEN RECEIVES A TRANSACTION-         */
            /* INDEPENDENT NTP ERROR CLASS CODE.             */
 3 TREXCEPTION_NUMBER FIXED BIN(15), /* ERROR NUMBER --       */
            /* SET TO NON-ZERO ONLY WHEN A TRANSACTION ERROR */
            /* OCCURS.  THEN RECEIVES A TRANSACTION-         */
            /* DEPENDENT ERROR NUMBER.                       */
 3 TRSTATE FIXED BIN(15),         /* TRANSACTION STATE --      */
            /* CONTAINS ONE OF THE CODES DEFINED BY          */
            /* INCLUDE PACKET PCSTATE.                       */
 3 TRID FIXED BIN(15),            /* INTERNAL NTP TRANS. ID.   */
```

(continued)

Figure 1 (continued): The PCTRANS Declaration

```
 3 TRTYPE CHAR(6),                  /* TRANSACTION TYPE:          */
            /* 'CALL', 'LOG', 'ASK', 'TELL', OR 'ALARM'.         */
 3 TRCOMPL CHAR(6),                 /* COMPLETION MODE:           */
            /* 'PEND', 'REPLY', 'ERROR', OR 'FREE'.              */
 3 TRLINK POINTER,                  /* INTERNAL CHAIN FIELD USED */
            /* BY PL/PCP TO HANG TOGETHER ALL TRANSACTIONS.      */
 3 TRHANDLING ALIGNED,              /* SPECIAL HANDLING REPORT    */
            /* FOR THE RECEIVED HALF OF THE TRANSACTION.         */
   5 (SEQUENCED, STREAM_MARKER) BIT(1) UNALIGNED,
 3 TRFLAGS BIT(8)                   /* PL/PCP INTERNAL FLAGS.  */
```

Figure 2: the PCSTATE Declaration

```
DECLARE
1 PCSTATE STATIC EXTERNAL,  /* ORDERED TRANSACTION STATES      */
 2 (TS_MESSAGE_SENT     INIT (1), /* THE TRANSACTION-CREATING  */
             /* MESSAGE HAS LEFT THE SENDER, BUT HAS NOT YET  */
             /* ARRIVED AT THE RECEIVER.                      */
    TS_MESSAGE_RECEIVED INIT (2), /* THE TRANSACTION-CREATING */
             /* MESSAGE HAS ARRIVED AT THE RECEIVER.          */
    TS_REPLY_SENT       INIT (3), /* THE TRANSACTION-COMPLETING*/
             /* MESSAGE HAS LEFT THE SENDER, BUT HAS NOT YET  */
             /* ARRIVED AT THE RECEIVER.                      */
    TS_COMPLETED        INIT (4)) /* THE TRANSACTION-COMPLETING*/
             /* MESSAGE HAS ARRIVED AT THE RECEIVER, OR THE   */
             /* TRANSACTION HAS BEEN ABANDONED DUE TO AN      */
             /* ERROR CONDITION.  IT IS TIME TO CALL PCFREE.  */
                          FIXED BIN(15);
```

Figure 3: The PCPROC Declaration

```
/* INTERNALS OF PROCESS STRUCTURE.  PROVIDE YOUR OWN LEVEL-1  */
/* DECLARATION AND TERMINATOR.                                */
 3 PRUSER_PTR POINTER,           /* A POINTER VARIABLE THAT   */
            /* BELONGS TO THE CALLER.  THIS IS PROVIDED      */
            /* ONLY AS A CONVENIENCE, IN CASE THE CALLER     */
            /* NEEDS TO CHAIN TO TABLES OF HIS OWN.          */
 3 PRUSER_INDEX FIXED BIN(31),   /* AN INTEGER VARIABLE THAT  */
            /* BELONGS TO THE CALLER.  THIS IS PROVIDED      */
            /* ONLY AS A CONVENIENCE, IN CASE THE CALLER     */
            /* NEEDS TO INDEX INTO TABLES OF HIS OWN.        */
 3 PRLINK POINTER,               /* INTERNAL CHAIN FIELD.     */
 3 PRRELATION FIXED BIN(15),     /* HOST TYPE OF THE PROCESS, */
            /*   0 = LOCAL, 1 = FAMILY, 2 = FOREIGN HOST.    */
 3 PRNAME,                       /* PL/MSG PROCESS NAME:      */
   5 PRHOST_ID FIXED BIN(15),    /* HOST NUMBER               */
   5 PRINCARNATION FIXED BIN(15),/* HOST MSG INCARNATION NO   */
   5 PRINSTANCE FIXED BIN(15),   /* PROCESS INSTANCE NO       */
   5 PRGENERIC CHAR(127) VAR,    /* PROCESS GENERIC NAME      */
 3 PRFLAGS,                      /* ATTRIBUTES:               */
   5 (PREXTROVERTED, PRDUMMY) BIT(1) ALIGNED,
 3 PRHOST CHAR(32) VAR,          /* MSGHTYP HOST NAME.        */
 3 PRTYPE CHAR(32) VAR,          /* MSGHTYP HOST FAMILY.      */
 3 PRPNAM CHAR(127) VAR          /* PNAMOUT PROCESS NAME.     */
```

5.3.10.2. PROCESS HANDLES

A remote process handle becomes defined when its value is returned from PCHELLO, from PCEXAM when it returns a type of "CALL" or "LOG", or from PCEXAM when it returns a type of "REPLY" or "ERROR" for a transaction created via PCCALL. The local process handle becomes defined when its value is returned from PCBEGIN. A remote process handle becomes undefined when it is passed to PCBYE, or when the local process dematerializes. The local process handle becomes undefined only when the local process dematerializes.

Besides its use as the name of a process, a process handle points to a PL/I BASED structure which contains some information of use to the caller. This structure is defined by %INCLUDE segment PCPROC (see figure 3).

5.4. BASIC HOUSEKEEPING IN PL/PCP

5.4.1. INITIALIZATION -- PCBEGIN

PCBEGIN must be called before any other PL/PCP or PL/MSG routine is called. It materializes the local MSG process, sets up operating parameters, defines the PL/PCP master ECB, and sets up the mechanisms for receiving messages and alarms. It does not ARM the process for alarms -- that is under the direct control of the process, via routine MSGAA.

If PCBEGIN is called a second time, it dematerializes the MSG process and materializes a new one. This may be repeated as many times as desired; however, only one process can be materialized at a time.

```
CALL PCBEGIN
     (generic name,              /* CHAR(*) VAR,
                                    You must set.     */
      message processing mode,/* FIXED BIN (15)
                                    You must set.     */
      caller recognition mode,/* BIT(1)
                                    You must set.     */
      msg timeout,               /* FIXED BIN (31)
                                    You must set.     */
      master ecb,                /* FIXED BIN (31)
                                    You must monitor. */
      gmt adjustment,            /* FIXED BIN (15,4)
                                    You must set.     */
      process handle);           /* POINTER
                                    PL/PCP returns.   */
```

where:

"generic name" declares the local process' MSG process class. It may not exceed 127 characters.

"message processing mode" identifies the process' behavior pattern as one of the following:

1 --> aggressive: this process will define its own procedure, and will never receive other than specifically addressed messages.

0 --> passive: this process will start by waiting for a generically-addressed remote CALL or LOG. After that, it will

never receive other than specifically addressed messages.

-1 --> recallable: this process will never receive other than generically addressed messages, and it may receive any number of them.

"caller recognition mode" identifies the process' behavior pattern as one of the following (the value of this argument is irrelevant for a recallable process):

0B --> extroverted: this process will accept specifically addressed CALLs or LOGs if and when they arrive.

1B --> introverted: this process will not accept specifically-addressed CALLs or LOGs (note that this does not prevent its functioning as a passive process).

"msg timeout" is the value to be used to time out all outgoing MSG messages (not transactions), in hundredths of real seconds. PL/PCP remembers the location of this datum, and its value is redetermined each time it is needed. This means that this datum should not be changed for the lifetime of the process, unless the intent is to alter MSG behavior.

"master ecb" is an ECB that PL/PCP will clear and then cause to be posted whenever an MSG elementary event occurs. The caller should never set or clear this word, but he must wait on it or test its completion bit. Whenever it is found posted, PCEXAM should be called immediately, to turn the elementary event into something meaningful in terms of transaction processing, and also to reenable any MSG pending event that was completed by the elementary event.

"gmt adjustment" is the number of hours earlier than Greenwich that the host operating system can be assumed to be operating. It carries four fractional bits to accomodate half-hour time zones. Typical values for common time zones would be: for EST, 5.0; for EDT, 4.0; for PST, 8.0; for PDT, 7.0.

"process handle" is a pointer returned to you by PCBEGIN. It is your handle to your local process identification. Most using programs will not need to preserve this value, since you never need to identify the local process to create and complete transactions. See the section entitled USING HANDLES for more details.

5.4.2. FREEING TRANSACTION HANDLES -- PCFREE

PL/PCP is constantly creating transaction structures and passing corresponding transaction handles back to the calling program. These structures must be freed when the caller is through with them, by calling routine PCFREE.

Normally, this is done when the transaction has completed, normally or abnormally, and there is no further possibility of events occurring that relate to it. However, any transaction can be freed at any time, if the intention is to abort it at its present state of processing. For example, it is common to call PCFREE to finalize a remote ALARM transaction to which no reply is to be sent. Transaction structures can be freed implicitly as a result of the PCBYE or PCEND routines. These are covered under the section entitled PL/PCP CALLS FOR SPECIAL SITUATIONS.

```
CALL PCFREE
     (transaction handle);     /* POINTER ,
                                  You must set.     */
```

where "transaction handle" contains the value associated with the transaction to be demolished.

5.5. SENDING MESSAGES WITH PL/PCP

There are six kinds of outgoing messages, and six corresponding PL/PCP entry points to send them. With the exception of PCALARM, whenever any of these routines sends a message, then if the process is operating under the PL/MSG run-time LOG option, routine PRINTB8 is called to print the outgoing message into the MSG journal.

5.5.1. CREATING A LOCAL CALL TRANSACTION -- PCCALL

The PCCALL routine creates a local CALL transaction and leaves it pending. The associated remote process will not become known until the transaction is completed, because all local CALLs use the MSG generic addressing feature.

```
CALL PCCALL
     (generic name,           /* CHAR(*) VAR,
                                 You must set.    */
      host,                   /* FIXED BIN(15)
                                 You must set.    */
      procedure,              /* CHAR(*) VAR
                                 You must set.    */
      arguments,              /* CHAR(*) VAR
                                 You must set.    */
      timeout,                /* FIXED BIN (31)
                                 You must set.    */
      wait enable,            /* BIT(1),
                                 You must set.    */
      transaction handle);    /* POINTER
                                 PL/PCP returns.  */
```

where:

"generic name" declares the remote process' MSG process class. It may not exceed 127 characters.

"host" is the NSW host number of the system where the process is to reside. A value of zero tells MSG to pick any host known to support the process class.

"procedure" is the EBCDIC name of the procedure to be called.

"arguments" is a NETB8 string containing the encoded sequence of procedure arguments, as generated by the PL/B8 package. Character strings need not have been converted to ASCII.

"timeout" is the transaction timeout interval, in hundredths of real seconds, or zero to disable timing.

"wait enable" is '0'B if you wish the creating message transmission to be rejected if the remote host does not already have a process waiting for a remote CALL or LOG. Otherwise, it is '1'B.

"transaction handle" is a pointer returned to you by PCCALL. It is your handle to your local transaction identification. You will eventually receive the same value back from the PCEXAM call that completes the transaction.

5.5.2. CREATING A LOCAL LOG TRANSACTION -- PCLOG

The PCLOG routine creates a local LOG transaction. No reply will be received; however, the transaction remains pending for the purpose of outgoing message error detection. The associated process will never become known because all local LOGs use the MSG generic addressing feature.

```
CALL PCLOG
     (generic name,          /* CHAR(*) VAR,
                                You must set.     */
      host,                  /* FIXED BIN(15)
                                You must set.     */
      procedure,             /* CHAR(*) VAR
                                You must set.     */
      arguments,             /* CHAR(*) VAR
                                You must set.     */
      wait enable,           /* BIT(1),
                                You must set.     */
      transaction handle);   /* POINTER
                                PL/PCP returns.   */
```

where:

"generic name" declares the remote process' MSG process class. It may not exceed 127 characters.

"host" is the NSW host number of the system where the process is to reside. A value of zero tells MSG to pick any host known to support the process class.

"procedure" is the EBCDIC name of the procedure to be called.

"arguments" is a NETB8 string containing the encoded sequence of procedure arguments, as generated by the PL/B8 package. Character strings need not have been converted to ASCII.

"wait enable" is '0'B if you wish the creating message transmission to be rejected if the remote host does not already have a process waiting for a remote CALL or LOG. Otherwise, it is '1'B.

"transaction handle" is a pointer returned to you by PCLOG. It is your handle to your local transaction identification. You will eventually receive the same value back from the PCEXAM call that completes the transaction.

5.5.3. CREATING A LOCAL ASK TRANSACTION -- PCASK

The PCASK routine creates a local ASK transaction directed toward a specific process and leaves it pending.

```
CALL PCASK
     (process handle          /* POINTER
                                 You must set.     */
      procedure,              /* CHAR(*) VAR
                                 You must set.     */
      arguments,              /* CHAR(*) VAR
                                 You must set.     */
      timeout,                /* FIXED BIN (31)
                                 You must set.     */
      special handling,       /* bit structure
                                 You must set.     */
      transaction handle);    /* POINTER
                                 PL/PCP returns.   */
```

where:

"process handle" defines the process to which the transaction is addressed. Its value will have been gotten from some previous call to a PL/PCP routine.

"procedure" is the EBCDIC name of the procedure to be called.

"arguments" is a NETB8 string containing the encoded sequence of procedure arguments, as generated by the PL/B8 package. Character strings need not have been converted to ASCII.

"timeout" is the transaction timeout interval, in hundredths of real seconds, or zero to disable timing.

"special handling" is a PL/PCP special handling code.

"transaction handle" is a pointer returned to you by PCASK. It is your handle to your local transaction identification. You will eventually receive the same value back from the PCEXAM call that completes the transaction.

5.5.4. CREATING A LOCAL TELL TRANSACTION -- PCTELL

The PCTELL routine creates a local TELL transaction directed toward a specific process. No reply will be received; however, the transaction remains pending for the purpose of outgoing message error detection.

```
CALL PCTELL
     (process handle          /* POINTER
                                 You must set.     */
      procedure,              /* CHAR(*) VAR
                                 You must set.     */
      arguments,              /* CHAR(*) VAR
                                 You must set.     */
      special handling,       /* bit structure
                                 You must set.     */
      transaction handle);    /* POINTER
                                 PL/PCP returns.   */
```

where:

"process handle" defines the process to which the transaction is addressed. Its value will have been gotten from some previous call to a PL/PCP routine.

"procedure" is the EBCDIC name of the procedure to be called.

"arguments" is a NETB8 string containing the encoded sequence of procedure arguments, as generated by the PL/B8 package. Character strings need not have been converted to ASCII.

"special handling" is a PL/PCP special handling code.

"transaction handle" is a pointer returned to you by PCTELL. It is your handle to your local transaction identification. You will eventually receive the same value back from the PCEXAM call that completes the transaction.

5.5.5. CREATING A LOCAL ALARM TRANSACTION -- PCALARM

The PCALARM routine creates a local ALARM transaction directed toward a specific process and leaves it pending.

```
CALL PCALARM
     (process handle          /* POINTER
                                 You must set.     */
      alarm code,             /* FIXED BIN(15)
                                 You must set.     */
      timeout,                /* FIXED BIN (31)
                                 You must set.     */
      transaction handle);    /* POINTER
                                 PL/PCP returns.   */
```

where:

"process handle" defines the process to which the alarm is addressed. Its value will have been gotten from some previous call to a PL/PCP routine.

"alarm code" is the binary code to be transmitted.

"timeout" is the transaction timeout interval, in hundredths of real seconds, or zero to disable timing.

"transaction handle" is a pointer returned to you by PCALARM. It is your handle to your local transaction identification. You will eventually receive the same value back from the PCEXAM call that completes the transaction.

5.5.6. COMPLETING A REMOTE TRANSACTION -- PCREPLY

The PCREPLY routine initiates the completion of a remote transaction. The transaction remains pending for the purpose of outgoing message error detection.

If any faults have been accumulated for the transaction, they will be collected into a fault descriptor and included in the reply message. In this case, the remote process will consider the transaction to complete in ERROR mode, even though the local process will consider it to complete normally.

```
CALL PCREPLY
     (transaction handle,     /* POINTER
                                 You must set.     */
      results,                /* CHAR(*) VAR
                                 you must set.     */
      special handling);      /* bit structure
                                 You must set.     */
```

where:

"transaction handle" identifies the remote transaction to be completed. It will have been gotten from the PCEXAM call that created the transaction. You will eventually receive the same value back from the PCEXAM call that actually completes the transaction.

"results" is a NETB8 REFER dope vector defining the encoded sequence of procedure results, as generated by the PL/B8 package. Character strings need not have been converted to ASCII.

"special handling" is a PL/PCP special handling code.

5.6. RECEIVING MESSAGES AND EVENTS WITH PL/PCP -- PCEXAM

There are seven kinds of incoming message types, corresponding to the six outgoing types listed above, but with two variants ("REPLY" and "ERROR") on transaction completion. There are also three non-message events associated with the message-receiving mechanism. A PL/PCP-using program does not know the type of an incoming message until it calls PL/PCP to have it examined; therefore, there is a single routine defined for this purpose. Its calling sequence is sufficiently general to accommodate all cases, through the artifice of returning results in a transaction structure rather than in parameter cells. What is returned, and what it means, varies according to the value returned in the "type" parameter.

PCEXAM should be called whenever the PL/PCP master ECB defined in the PCBEGIN call is found posted. Remember that this ECB stands in for a variable set of elementary events that PL/PCP is keeping track of for you, and that a single posting of it may represent the completion of more than one such event. Only one event can be reported by a single call to PCEXAM, so if more events are complete, the ECB will be left posted when PCEXAM returns to you. For this reason, you must <u>never</u> clear the ECB yourself once you have called PCBEGIN. In selecting which event to return to you on a single call, PCEXAM will give first priority to alarms.

Notice that if this call results in the creation or completion of a transaction, the caller is not necessarily obliged to do more than make a note of that fact. Thus he should not wait until he can conveniently process such an event before calling PCEXAM.

Whenever PCEXAM receives a message of any type (except an alarm), then if the process is operating under the PL/MSG run-time LOG option, routine PRINTB8 is called to print the outgoing message into the MSG journal.

The general form of the call is:

```
CALL PCEXAM
     (type,                    /* CHAR(6)
                                  PL/PCP returns.   */
      transaction handle);     /* POINTER
                                  PL/PCP returns.   */
```

where:

"type" will receive one of the seven message types, or one of the three non-message everts: "NULL", "FREE", or "TERM".

"transaction handle" will receive the value assiged to the transaction of which the newly-arrived message is a part, or will be NULL for a non-message event that is not associated with a transaction.

5.6.1. RECEIVING A REMOTE CALL TRANSACTION

You are notified of the creation of a remote CALL transaction by a "type" of "CALL" returned from PCEXAM. The transaction remains pending. A type of "CALL" will never be returned from PCEXAM for an agressive introverted process. For a passive introverted process, only the first message-bearing transaction can be a "CALL". For an extroverted or recallable process, "CALL" can be returned at any time.

When "CALL" is returned, "transaction handle" will point to a new transaction structure in which are returned:

TRPROCESS contains the process handle assigned the newly recognized process.

TRPROCEDURE points to the EBCDIC name of the procedure being called.

TRARGUMENTS is a "SEQL" dope vector that now defines the still-encoded sequence of procedure arguments. All contained character strings will be in EBCDIC.

TRHANDLING is set to indicate whether any MSG special handling options are in effect for the message.

5.6.2. RECEIVING A REMOTE LOG TRANSACTION

You are notified of the creation of a remote LOG transaction by a "type" of "LOG" returned from PCEXAM. A new transaction has been both defined and completed, so it does not remain pending. A type of "LOG" will never be returned from PCEXAM for an agressive introverted process. For a passive introverted process, only the first message-bearing transaction can be a "LOG". For an extroverted or recallable process, "LOG" can be returned at any time.

When "LOG" is returned, "transaction handle" will point to a new transaction structure in which are returned:

TRPROCESS contains the process handle assigned the newly recognized process.

TRPROCEDURE points to the EBCDIC name of the procedure being called.

TRARGUMENTS is a "SEQL" dope vector that now defines the still-encoded sequence of procedure arguments. All contained character strings will be in EBCDIC.

TRHANDLING is set to indicate whether any MSG special handling options are in effect for the message.

5.6.3. RECEIVING A REMOTE ASK TRANSACTION

You are notified of the creation of a remote ASK transaction by a "type" of "ASK" returned from PCEXAM. The transaction remains pending.

When "ASK" is returned, "transaction handle" will point to a new transaction structure in which are returned:

TRPROCESS contains the process handle previously assigned this process.

TRPROCEDURE points to the EBCDIC name of the procedure being called.

TRARGUMENTS is a "SEQL" dope vector that now defines the still-encoded sequence of procedure arguments. All contained character strings will be in EBCDIC.

TRHANDLING is set to indicate whether any MSG special handling options are in effect for the message.

5.6.4. RECEIVING A REMOTE TELL TRANSACTION

You are notified of the creation of a remote TELL transaction by a "type" of "TELL" returned from PCEXAM. A new transaction has become defined and completed, so it does not remain pending.

When "TELL" is returned, "transaction handle" will point to a new transaction structure in which are returned:

TRPROCESS contains the process handle previously assigned this process.

TRPROCEDURE points to the EBCDIC name of the procedure being called.

TRARGUMENTS is a "SEQL" dope vector that now defines the still-encoded sequence of procedure arguments. All contained character strings will be in EBCDIC.

TRHANDLING is set to indicate whether any MSG special handling options are in effect for the message.

5.6.5. RECEIVING A REMOTE ALARM TRANSACTION

You are notified of the creation of a remote ALARM transaction by a "type" of "ALARM" returned from PCEXAM. The transaction remains pending.

When "ALARM" is returned, "transaction handle" will point to a new transaction structure in which are returned:

TRPROCESS contains the process handle previously assigned this process.

TRID contains the recieved alarm code.

5.6.6. REPLY COMPLETION OF A LOCAL TRANSACTION

You are notified of the normal completion of a local transaction by a "type" of "REPLY" returned from PCEXAM.

When "REPLY" is returned, "transaction handle" will point to a previously created transaction structure in which are returned:

TRPROCESS contains the process handle value assigned this remote process. If the local transaction being completed was created via PCCALL, this represents a newly-recognized process. Otherwise, it is the same value that the caller used to create the transaction.

TRARGUMENTS is a "SEQL" dope vector that now defines the still-encoded sequence of procedure results. All contained character strings will be in EBCDIC.

TRHANDLING is set to indicate whether any MSG special handling options are in effect for the message.

5.6.7. ERROR COMPLETION OF A LOCAL TRANSACTION

You are notified of the abnormal completion of a local transaction by a "type" of "ERROR" returned from PCEXAM.

When "ERROR" is returned, "transaction handle" will point to a previously created transaction structure in which are returned:

TRPROCESS contains the process handle value assigned this remote process. If the local transaction being completed was created via PCCALL, this represents a newly-recognized process. Otherwise, it is the same value that the caller used to create the transaction.

TREXCEPTION_CLASS contains a code that identifies the basic level of error that has occurred. There are three cases:

* Code < 0 -- The error is on the PCP level. At this writing, the only such error is a PCP transaction timeout.

* Code > 0 -- The error is on the MSG level. The value of the code is a PL/MSG error code (see reference 5).

* Code = 0 -- The error is on the called procedure level. There is a fault descriptor set associated with the transaction, and it should be retrieved through routine PCGETF.

TRARGUMENTS is a "SEQL" dope vector that now defines the still-encoded sequence of procedure results. All contained character strings will be in EBCDIC.

TRHANDLING is set to indicate whether any MSG special handling options are in effect for the message.

5.6.8. RECEIVING NON-MESSAGE EVENTS

There are three kinds of non-message events that can be discovered by PCEXAM.

5.6.8.1. THE FREE EVENT

When PCEXAM returns a type of "FREE", "transaction handle" points to a transaction block representing a remote transaction that has been completed by the using process. This indicates that the message completing the transaction has been succesfully transmitted (If this were not the case, a type of "ERROR" would be returned). The transaction should be freed through routine PCFREE.

5.6.8.2. THE NULL EVENT

When PCEXAM returns a type of "NULL", "transaction handle" will contain the PL/I value NULL. By a NULL type event, we mean an elementary event that does not cause the completion of any transaction event. Because of the wait structure of PL/PCP, this kind of message will be common. You must code every call to PCEXAM such that if a type of "NULL" is returned, you can continue processing (or waiting) just as if the elementary event in question had not occurred.

5.6.8.3. THE TERM EVENT

When PCEXAM returns a type of "TERM", "transaction handle" will contain the PL/I value NULL. A TERM type event represents MSG's having posted the process' terminationsignal ECB. The process should complete or abandon its current activity, clean up as rapidly as possible, and dematerialize.

5.7. FAULT HANDLING IN PL/PCP

A PL/PCP transaction may have associated with it a set of faults. Each fault represents a specific exceptional condition that arose during the processing of the transaction.

Faults for a local transaction are created by the remote process, and become available to the local process when the transaction completes in ERROR mode. The local process may choose to ignore them or examine them with the PCGETF routine.

Faults for a remote transaction are usually created by the local process and assigned the transaction by the PCPUTF routine. They can also be copied from another completed transaction with the PCCOPYF routine.

The components of a fault are:

* The "fault class" is a procedure-independent fault code that can be interpreted by any NSW process. The values of this code will be specified by NSW designers.

* The "fault number" is a procedure-dependent error code that is defined in terms of the procedure that was invoked by the transaction.

* The "fault string" is a short description of the fault in terms a user can understand.

* The "debug report" is a list of debugging information which is mostly handled internal to PL/PCP and related software packages.

* There are other components of a fault, referred to in NSW documentation (see reference 3) as the "fault function" and the "faultstamp". These components have not yet been integrated into the PL/PCP design.

The transaction structure based upon a transaction handle contains two "SEQL" dope vectors related to fault processing. The first of these, TRFAULTS, defines the current fault set for the transaction. Subfield TRFCOUNT tells how many members are in that set. The second dope vector, TRFAULTSCAN, describes that subset of the fault set that remains to be examined through routine PCGETF. Subfield TRSCOUNT tells how may members are in that subset. Notice that TRFAULTSCAN is not well maintained when PCPUTF or PCCOPYF is called against a transaction, since it is not expected that either of these routines will be used against the same transaction as PCGETF.

5.7.1. EXAMINING INCOMING FAULTS -- PCGETF

The PCGETF routine is used repeatedly to examine each member of the set of faults attached to a local transaction that has completed in ERROR mode. Each time it is called it updates TRFAULTSCAN (initialized by PCEXAM) in such a way that each fault is produced in turn until TRSCOUNT becomes zero. The caller should use the value of TRSCOUNT to terminate his calling loop; however, calling PCGETF with all faults examined merely returns "null" values, and causes no harm.

```
CALL PCGETF
     (transaction handle,     /* POINTER
                                 You must set.     */
      fault class,            /* FIXED BIN(15)
                                 PL/PCP returns.   */
      fault number,           /* FIXED BIN (15)
                                 PL/PCP returns.   */
      fault string,           /* CHAR(*) VAR
                                 PL/PCP returns.   */
      debug descriptor);      /* "SEQL" dope vector
                                 PL/PCP returns.   */
```

where:

"transaction handle" identifies the transaction whose faults are to be examined.

"fault class", "fault number", and "Fault string" are variables to receive the corresponding data values from the fault being examined. The string may be truncated if necessary.

"debug descriptor" will receive values describing the fault's debug list. If the program is interested in this data, it must parse and examine it on its own. Notice that if the PL/MSG LOG option is enabled, all this data will already have been printed into the MSG journals, and that this may be sufficient.

5.7.2. ACCUMULATING OUTGOING FAULTS -- PCPUTF

The PCPUTF routine is used to add a single fault to the set that is being accumulated during the processing of a remote transaction. It updates TRFAULTS and copies it into TRFAULTSCAN.

```
CALL PCPUTF
     (transaction handle,     /* POINTER
                                 You must set.     */
      fault class,            /* FIXED BIN(15)
                                 You must set.     */
      fault number,           /* FIXED BIN (15)
                                 You must set.     */
      fault string,           /* CHAR(*) VAR
                                 You must set.     */
      debug descriptor);      /* "SEQL" dope vector
                                 You must set.     */
```

where:

"transaction handle" identifies the transaction which is to receive the fault.

"fault class", "fault number", and "fault string" are variables containing the corresponding data values for the fault.

"debug descriptor" is normally a null string, indicating that PL/PCP is to use its own judgement in building a debug list for the fault. Alternatively, it can be a "NETB8" string containing a complete debug descriptor built by the caller.

5.7.3. PROPAGATING FAULTS -- PCCOPYF

The PCCOPYF routine is used to copy the entire set of faults attached to a completed local transaction into the set being accumulated during the processing of a remote transaction of which the completed transaction is a part. For the latter transaction, it updates TRFAULTS and copies it into TRFAULTSCAN.

```
CALL PCCOPYF
     (input transaction,      /* POINTER
                                 You must set.     */
      output transaction);    /* POINTER
                                 You must set.     */
```

where:

"input transaction" is the handle of the transaction which is to be the source of the copy.

"output transaction" is the handle of the transaction which is to receive the fault set.

5.7.4. AUTOMATIC FAULT LOGGING

Whenever any PL/PCP routine sends or receives a message containing a non-null fault descriptor, it will attempt to create and complete a hidden LOG transaction which sends the fault descriptor to one or more NSW error logging processes. The local process will never be aware of this transaction, even if it encounters faults of its own. PL/PCP is protected against recursion from such second-level faults.

5.8. PL/PCP CALLS FOR SPECIAL SITUATIONS

The PL/PCP features discussed in this section are not needed by using programs unless special situations exist.

5.8.1. MEETING A NEW PROCESS OUTSIDE PL/PCP -- PCHELLO

The using program normally learns about a remote process when a PL/PCP routine returns it a new process handle. However, it is legitimate for the local process to learn the specific name of a second process through the data portion of a message from a third process. In this case, the using program must specifically tell PL/PCP about the new process before conversing with it. The PCHELLO routine does this:

```
CALL PCHELLO
     (process name            /* NSWB8 "PROCNM" descr.
                                 You must set.     */
      process handle);        /* POINTER
                                 PL/PCP returns.   */
```

where:

"process name" is a PL/B8 MSG process name (map string type "PROCNM") descriptor, exactly as returned from the PL/B8 routine that created it.

"process handle" is a PL/PCP process handle which can represent the process in subsequent PL/PCP calls.

5.8.2. BREAKING OFF WITH A REMOTE PROCESS -- PCBYE

Sometimes a local process wants to forget about one of its known remote processes. The PCBYE routine does this:

```
CALL PCBYE
     (process handle);        /* POINTER
                                 You must set.     */
```

On return, the referenced process structure has been freed, the current value of this handle has become undefined, and PCFREE has been called for any transaction structures associated with the process.

5.8.3. DEMATERIALIZING THE LOCAL PROCESS -- PCEND

The local process may be dematerialized in three ways. If the using task terminates, the process is automatically dematerialized. If PCBEGIN is called to materialize a new process, the old one is first dematerialized. You can also dematerialize the process by a call of the form:

```
CALL PCEND;
```

On return from this routine, the caller is free to materialize another process, to terminate, or to continue execution dealing with non-MSG work. If any process or transaction structures were still defined, they will have been freed.

5.8.4. READING THE PCP CLOCK -- PCTOD

The PCTOD routine provides a standard timestamp for the internal routines of PL/PCP; however, it can be used by any caller, using the form:

```
timestamp =                    /* FLOAT BIN(44)
   PCTOD                          PL/PCP returns.   */
     (gmt adjustment);         /* FIXED BIN(15,4)
                                    You must set.     */
```

where:

"gmt adjustment" is the same as the similar parameter to PCBEGIN.

"timestamp" is the current date and time, in the standard form:

* bits 00-07: exponent -- always X'4E'.
* bits 08-31: day number -- the number of days since November 17, 1858.
* bits 32-49: integer part of the time of day, in seconds since midnight.
* bits 50-63: fractional part of the time of day.

These fields are arranged so that bits 21-52 are the NSWNT standard timestamp.

5.8.5. STORING STRINGS WITH A TRANSACTION -- PCTEXT

Every transaction has associated with it a chain of buffers containing the message components pointed to by the various fields of the transaction structure. These buffers are built and maintained by various PL/PCP routines, and are freed by PCFREE. Usually, a using program need not be aware of them. However, if it is ever helpful for a using program to add buffers to the chain, the PCTEXT routine can be used:

```
text handle =                    /* POINTER
   PCTEXT                           PL/PCP returns.   */
     (transaction handle,        /* POINTER
                                     You must set.     */
```

where:

"transaction handle" identifies the transaction with which the text buffer is to be associated.

"character string" is a varying string containing the text to be stored in the buffer.

"text handle" receives the address of a buffer of the form described by the PCTBUF %INCLUDE segment (see figure 4). The current length and value of the stored string will be those of "character string"; however, its maximum length will be set to its current length. This handle will be valid as long as the associated transaction handle remains valid.

Figure 4: The PCTBUF Declaration

```
/* THE TRANSACTION TEXT BUFFER STORED BY ROUTINE PCTEXT.      */
/* PROVIDE YOUR·OWN LEVEL-1 DECLARATION AND TERMINATOR.       */
 3 TTLINK POINTER,                /* INTERNAL BUFFER CHAIN.   */
 3 TTLENGTH FIXED BIN (15),       /* LENGTH OF TTEXT FIELD.   */
 3 TTEXT CHAR (1 REFER (TTLENGTH)) VAR   /* ACTUAL TEXT.      */
```

5.8.6. BUILDING A DUMMY TRANSACTION -- PCFAKET

The PCFAKET routine provides the caller with a way to build a dummy transaction structure, for whatever purposes he wishes (e. g., as a base for storing strings). It is called by:

```
transaction handle =          /* POINTER
   PCFAKET;                      PL/PCP returns.   */
```

where:

"transaction handle" is the handle of the dummy transaction structure.

5.9. COMPATIBILITY WITH VERSION 1 -- PCERROR

The PCERROR routine is included for compatibility with previous versions of PL/PCP. Its functions have now been taken over by PCPUTF and PCREPLY. It should not be used in new programming. This routine initiates the completion of a remote transaction in ERROR mode. The transaction remains pending for the purpose of outgoing message error detection.

The caller specifies a particular fault to be included in the reply message. However, if other faults have been accumulated for the transaction, this one will merely be added to the collection, which will then be formatted into a fault descriptor and included in the reply message.

```
CALL PCERROR
     (transaction handle,     /* POINTER
                                 You must set.     */
      results,                /* CHAR(*) VAR
                                 you must set.     */
      fault string,           /* CHAR(*) VAR,
                                 You must set.     */
      fault class,            /* FIXED BIN(15)
                                 You must set.     */
      fault number,           /* FIXED BIN(15)
                                 You must set.     */
      special handling);      /* bit structure
                                 You must set.     */
```

where:

"transaction handle" identifies the remote transaction to be completed. It will have been gotten from the PCEXAM call that created the transaction. You will eventually receive the same value back from the PCEXAM call that actually completes the transaction.

"results" is a NETB8 REFER dope vector defining the encoded sequence of procedure results, as generated by the PL/B8 package. Character strings need not have been converted to ASCII.

"fault string", "fault class", and "fault number" are as described in the section entitled FAULT HANDLING IN PL/PCP.

"special handling" is a PL/PCP special handling code.

5.10. POSSIBLE TRANSACTION STATE CHANGES

In order to make absolutely clear the situations with which a PL/PCP caller must be prepared to deal, this section will detail the possible state changes for each transaction type. This information is presented in program-like structures; however, these are not necessarily similar to the program structures required to process transactions. They are intended only as convenient textual representations of state-change graphs.

In each case, the graph begins with the transaction-creating event and shows every legal path to transaction destruction, which is always accomplished by PCFREE. Constructs of the form <a,b,c> represent settings of TRTYPE, TRCOMPL, and TRSTATE, respectively.

* LOCAL CALL TRANSACTIONS

```
on PCCALL: <CALL,PEND,TS_MESSAGE_SEPC>
1) on PCFREE: <>
2) on PCEXAM(ERROR): <CALL,ERROR,TS_COMPLETED>
   on PCFREE: <>
3) on PCEXAM(NULL): <CALL,PEND,TS_MESSAGE_RECEIVED>
   3.1) on PCFREE: <>
   3.2) on PCEXAM(REPLY): <CALL,REPLY,TS_COMPLETED>
        on PCFREE <>
   3.3) on PCEXAM(ERROR): <CALL,ERROR,TS_COMPLETED>
        on PCFREE <>
```

* LOCAL LOG TRANSACTIONS

```
on PCLOG: <LOG,PEND,TS_MESSAGE_SEPC>
1) on PCFREE: <>
2) on PCEXAM(ERROR): <LOG,ERROR,TS_COMPLETED>
   on PCFREE: <>
3) on PCEXAM(FREE): <LOG,FREE,TS_COMPLETED>
   on PCFREE: <>
```

```
* LOCAL ASK TRANSACTIONS

     on PCASK: <ASK,PEND,TS_MESSAGE_SEPC>
     1) on PCFREE: <>
     2) on PCEXAM(ERROR): <ASK,ERROR,TS_COMPLETED>
        on PCFREE: <>
     3) on PCEXAM(NULL): <ASK,PEND,TS_MESSAGE_RECEIVED>
        3.1) on PCFREE: <>
        3.2) on PCEXAM(REPLY): <ASK,REPLY,TS_COMPLETED>
             on PCFREE <>
        3.2) on PCEXAM(ERROR): <ASK,ERROR,TS_COMPLETED>
             on PCFREE <>


* LOCAL TELL  TRANSACTIONS

     on PCTELL: <TELL,PEND,TS_MESSAGE_SEPC>
     1) on PCFREE: <>
     2) on PCEXAM(ERROR): <TELL,ERROR,TS_COMPLETED>
        on PCFREE: <>
     3) on PCEXAM(FREE): <TELL,FREE,TS_COMPLETED>
        on PCFREE: <>


* LOCAL ALARM TRANSACTIONS

     on PCALARM: <ALARM,PEND,TS_MESSAGE_SEPC>
     1) on PCFREE: <>
     2) on PCEXAM(ERROR): <ALARM,ERROR,TS_COMPLETED>
        on PCFREE: <>
     3) on PCEXAM(NULL): <ALARM,PEND,TS_MESSAGE_RECEIVED>
        3.1) on PCFREE: <>
        3.2) on PCEXAM(REPLY): <ALARM,REPLY,TS_COMPLETED>
             on PCFREE <>
        3.2) on PCEXAM(ERROR): <ALARM,ERROR,TS_COMPLETED>
             on PCFREE <>
```

```
* REMOTE CALL TRANSACTIONS

     on PCEXAM(CALL): <CALL,PEND,TS_MESSAGE_RECEIVED>
     1) on PCFREE: <>
     2) on PCREPLY: <CALL,REPLY,TS_REPLY_SEPC>
        2.1) on PCEXAM(FREE): <CALL,REPLY,TS_COMPLETED>
             on PCFREE: <>
        2.2) on PCEXAM(ERROR):  <CALL,ERROR,TS_COMPLETED>
             on PCFREE: <>

* REMOTE LOG TRANSACTIONS

     on PCEXAM(LOG): <LOG,FREE,TS_COMPLETED>
     on PCFREE: <>

* REMOTE ASK TRANSACTIONS

     on PCEXAM(ASK): <ASK,PEND,TS_MESSAGE_RECEIVED>
     1) on PCFREE: <>
     2) on PCREPLY: <ASK,REPLY,TS_REPLY_SEPC>
        2.1) on PCEXAM(FREE): <ASK,REPLY,TS_COMPLETED>
             on PCFREE: <>
        2.2) on PCEXAM(ERROR):  <ASK,ERROR,TS_COMPLETED>
             on PCFREE: <>

* REMOTE TELL TRANSACTIONS

     on PCEXAM(TELL): <TELL,FREE,TS_COMPLETED>
     on PCFREE: <>

* REMOTE ALARM TRANSACTIONS

     on PCEXAM(ALARM): <ALARM,PEND,TS_MESSAGE_RECEIVED>
     1) on PCFREE: <>
     2) on PCREPLY: <ALARM,REPLY,TS_REPLY_SEPC>
        2.1) on PCEXAM(FREE): <ALARM,REPLY,TS_COMPLETED>
             on PCFREE: <>
        2.2) on PCEXAM(ERROR):  <ALARM,ERROR,TS_COMPLETED>
             on PCFREE: <>
```

5.11. CANNED PL/I DECLARATIONS

For the convenience of the PL/I programmer, certain declarations will be stored in a public library. These can be invoked through the PL/I %INCLUDE statement. Each PL/PCP entry declaration has been stored under its own name, and various useful stuctures are also declared. The available names are:

PCALARM
PCASK
PCBEGIN
PCBYE
PCCALL
PCCOPYF
PCEND
PCERROR
PCEXAM
PCFREE
PCGETF
PCHELLO
PCLOG
PCPROC (the process strucure)
PCPUTF
PCREPLY
PCSTATE (the transaction state codes)
PCTBUF (the text buffer format)
PCTELL
PCTIME
PCTRANS (the transaction strucure)

5.12. STATUS OF THE IMPLEMENTATION

This document describes version 2 of the PL/PCP package. At this writing, version 2 is only partially implemented, and thus some parts of this document do not yet apply. Only the reader who is concerned with programming using PL/PCP now should read this section. Notable omissions in the version 2 implementation are:

* TIMEOUTS -- No transactions ever time out in response to the PL/PCP timeout values passed to routines that create local transactions. PL/MSG timeouts are fully functional.

* FAULT HANDLING -- The only mechanism for fault handling is the PCERROR routine, which is documented here in a section entitled "COMPATIBILITY WITH VERSION 1 -- PCERROR". PCGETF, PCPUTF, and PCCOPYF are unimplemented. The TRFAULTS and TRFAULTSCAN fields of the transaction structure are not implemented. When a type of ERROR is returned from PCEXAM, the exception data are returned in an manner incompatible with this body of this document, namely:

  TRPROCEDURE points to a human-readable EBCDIC description of the error.

  TREXCEPTION_CLASS is a procedure-independent error code that can be interpreted by any NSW process. Non-negative values are NSW error classes. Negative values are from PL/PCP, with these values currently defined:

  -1 means that a local transaction has been aborted due to an MSG timeout when sending the transaction-creating message.

  -2 means that a local transaction has been aborted due to a PCP timeout when awaiting a reply from the remote process. At this writing, this code is not yet being returned (see above).

  -3 means that the transaction has been aborted due to an error that defies PL/PCP analysis.

  TREXCEPTION_NUMBER is a procedure-dependent error code that is defined in terms of the procedure that was invoked by the transaction. If TREXCEPTION_CLASS is negative, this is zero.

* ALARM HANDLING -- An incoming alarm code is presently returned in TREXCEPTION_CODE, not in TRID as should be the case.

* AUTOMATIC FAULT LOGGING -- The mechanism for automatic fault logging is unimplemented.

* TIMESTAMPS -- The NSW standard timestamp as supported by PL/PCP is implemented as stated in this document; however, the latest NSW specification of this datum (see reference 3) uses a different format.

REFERENCES

[1] Schantz and Millstein, "The Foreman: Providing the Program Execution Environment for the National Software Works". Massachusetts Computer Associates Report No. CADD-7701-0111, January 1, 1977.

[1] NSW Protocol Committee, "MSG: The Interprocess Communication Facility for the National Software Works". Massachusetts Computer Associates Report No. CADD-7601-2611, January 23, 1976.

[3] Ulmer and Schantz, "NSW Fault Logger". BBN NSW Note 25, not dated.

[4] Braden and Ludlam, "PL/B8 -- A PL/I Interface for NSWB8". UCLA document UCNSW-403, November 15, 1980.

[5] Ludlam and Rivas, "PL/MSG -- An MSG Interface for PL/I". UCLA document UCNSW-401, November 15, 1980.

PART VI

PLIDAIR -- DYNAMIC ALLOCATION FROM PL/I

This section is separately available
as UCLA document UCNSW-404

6. PART VI: THE PLIDAIR PACKAGE

6.1. INTRODUCTION

PLIDAIR is a collection of PL/I-callable assembly language routines providing, in a succinct and simplified form, the basic services of TSO's Dynamic Allocation Interface Routine (DAIR) [ref. 1], some of the Catalog Management services [ref. 2], and optimum block size calculation, at data set creation, thru CCNDEVTP [ref. 3]. Optionally, it also provides an event journaling facility to keep a trace of all service requests and their results.

By allowing the syntax of data set names to optionally include "member names" most of the dynamic allocation and catalog management services have been simply and transparently extended to apply to the management of Partitioned Data set (PDS) members.

In its present state PLIDAIR is a collection of reentrant entries within a single Assembly-language Control Section named PLIDAIR. The package occupies about 6200 (decimal) bytes. The entries use the parameter-passing conventions defined by the IBM PL/I optimizing compiler [ref. 4], for external entries that are NOT declared with an OPTIONS clause.

6.2. SERVICES

The following entry points and their corresponding services are provided:

CREATE -- creates a brand new data set according to parameterized characteristics.

ALLOC -- binds a file name to an old data set or to a partitioned data set member.

ALLOCK -- performs the same function as ALLOC, except that ALLOCK of a file name to a partitioned data set member ensures that the member does exist, else an error return takes place.

CREALL -- creates a new data set according to given characteristics, and then allocates the given file name to it; it is the same as doing a CREATE followed by an ALLOC except that the RLSE option is set.

FREEDDN -- unbinds the given filename from whichever data set or PDS member it is currently allocated to.

FREEDSN -- unbinds the given data set or PDS member from whichever filename it is currently allocated to.

DELETE -- deletes an allocated or unallocated data set or PDS member.

FINDDSN -- locates and returns the data set name or PDS member name currently allocated to the given filename.

DELINDX -- deletes from the catalog the given fully qualified index.

RENAME -- renames data sets and PDS members.

6.3. PARAMETERS

Most of the generic parameters used to invoke the dynamic allocation and catalog management services are straightforward except for dsnames. These follow TSO syntax [ref. 6], and thus come in two forms: qualified and unqualified. The former must be enclosed in single quotes to denote qualification, while the latter must be unquoted to denote lack of qualification. Unquoted data set names are automatically qualified by PLIDAIR, while quoted ones are not. Partitioned data set member names are simply specified by suffixing to the partitioned data set name the member name enclosed in parentheses. A single unquoted asterisk stands in as the dsname associated with the TSO terminal. These are examples of legal dsnames:

```
OBJECT                      (unqualified data set)
'AHA323.NWT.OBJECT'         (qualified data set)
OBJECT(ABC)                 (unqualified member)
'AHA323.NWT.OBJECT(ABC)'    (qualified member)
*                           (terminal)
```

By convention, the last parameter to each entry is a return code variable (however, this convention is violated by the routines ALLOC and ALLOCK when their final optional volume parameter is used). On return from any routine, the return code variable contains a code selected from this set:

0 - uneventful call
1 - duplicate dsname or ddname
2 - dsname or ddname not found
3 - some resource is not available
4 - any other error
5 - inconsistent data structure

Following is a complete summary of the generic parameter forms required to interface to the various PLIDAIR services. All parameter types labeled "I" are of the input type, while those labeled "O" are of the output type. Exceptions are: the "dsn" parameter for FINDDSN is of type "O", while the "ddn" parameter for ALLOC, ALLOCK, and CREALL when set to blanks becomes an "O" parameter in which a system-generated file name is returned.

```
I flag -- BIT(1) ALIGNED -- Old flag:  0=shr, 1=old.

O rc   -- FIXED BIN(15) -- PLIDAIR return code.

O bsiz -- FIXED BIN(15) -- Optimum block size as calculated
          by CREATE and CREALL.   Set on return from those
          routines.

I vol  -- CHAR(6) VAR -- Volume serial label.

I ddn  -- CHAR(8) VAR -- File name.

I dsn  -- CHAR(56) VAR -- Data set or PDS member name,
          qualified or unqualified.

I old  -- CHAR(56) VAR -- Old data set name, like "dsn".

I new  -- CHAR(56) VAR -- New data set name, like "dsn".

I indx -- CHAR(56) VAR -- index name, like "dsn", but must
          be fully qualified (and therefore quoted).

I psd  -- POINTER -- locator of the following structure,
          containing the parameters required by CREATE and
          CREALL to create brand new data sets.

     1 physical_structure_descriptor,
       2 dsorg  CHAR(6) VAR,    /* PS, PO, DA, or IS      */
       2 recfm  CHAR(6) VAR,    /* any valid combination of
                                    F,V,U,B,T,A,M,S        */
       2 optcd  CHAR(6) VAR,    /* any valid combination of
                                    W,C,Q,T                */
       2 lrecl  FIXED BIN(15), /* logical record length  */
       2 maxbl  FIXED BIN(15), /* maximum block size     */
       2 minbl  FIXED BIN(15), /* minimum block size     */
       2 keylen FIXED BIN(15), /* key length             */
       2 rkp    FIXED BIN(15), /* relative key position  */
       2 palloc FIXED BIN(15), /* primary space allocation
                                                 in blocks */
       2 salloc FIXED BIN(15), /* secondary allocation   */
       2 dalloc FIXED BIN(15); /* directory allocation
                                  in 256-byte blocks     */
```

6.4. CALLING SEQUENCES

The calling sequences to the PLIDAIR services are given below. The parameters are as defined in the previous section. Notice that the declarations of these entries should NOT include the "OPTIONS (ASSEMBLER)" clause.

Create a data set (note 4)

```
CALL CREATE (dsn,vol,psd,bsiz,rc);
```

Allocate a File (notes 1,2,3,8)

```
CALL ALLOC (ddn,dsn,flag,rc,/'vol/");
```

Allocate and Check (notes 1,2,3,8)

```
CALL ALLOCK (ddn,dsn,flag,rc,/'vol/");
```

Create and Allocate (notes 3,4)

```
CALL CREALL (ddn,dsn,vol,psd,bsiz,rc);
```

Free a File

```
CALL FREEDDN (ddn,rc);
```

Free a Data Set

```
CALL FREEDDSN (dsn,rc);
```

Delete a Data Set

```
CALL DELETE (dsn,rc);
```

Find an Allocation (note 5)

```
CALL FINDDSN (ddn,dsn,rc);
```

Delete Index (notes 6,7)

```
CALL DELINDX (indx,rc);
```

Rename a Data Set

```
CALL RENAME (old,new,rc);
```

Note 1 -- When allocating a file to a PDS member, ALLOCK performs the allocation if and only if the PDS member actually exists, while ALLOC performs the allocation regardless of whether the member exists or not. ALLOC, when followed by the usual OPEN, output operations, and CLOSE, is the way to create new members and add their names to the PDS directory.

Note 2 -- To allocate a file to the terminal, use "*" for the data set name, per usual TSO conventions.

Note 3 -- When performing an ALLOC/ALLOCK or CREALL operation, if the ddn name provided is all blanks, then on return "ddn" is set to a system generated DD name.

Note 4 -- For both CREATE and CREALL, on return "bsiz" contains the actual block size chosen at data-set creation time by CCNDEVTP [ref. 3], the optimal blocksize calculator.

Note 5 -- On successful return from FINDDSN, "dsn" contains the fully qualified data set name currently allocated to the given "ddn" parameter. If the file is currently allocated to the terminal then the "dsn" returned is simply "*".

Note 6 -- The delete index service DELINDX deletes only the innnermost index level of the given fully qualified index. Thus if "indx" is "'AHA323.NWT.NSW.CCN'" then the index level "CCN" is deleted. More indices can be deleted thru programmed iteration by the user.

Note 7 -- Unlike most entries, DELINDX requires that the index name be fully qualified (and therefore quoted).

Note 8 -- The "vol" parameter to ALLOC and ALLOCK is optional. If it is not present, the data set is located through the system catalog, which is the preferred way.

6.5. JOURNAL MESSAGES

As previously mentioned, service requests, completion codes, and abnormal conditions are fully reported in the form of messages suitably formatted for writing to a journal log. To make use of this facility the load module containing PLIDAIR should include an entry named "MSGJOUR", which will be called whenever PLIDAIR has a message to write. If no such entry is present in the containing load module, journalling is bypassed. PLIDAIR uses a "weak reference" to MSGJOUR, so its absence does not cause a Linkage-Editor error.

If MSGJOUR is present, its usage conventions must be:

```
DCL MSGJOUR ENTRY(CHAR(*) VAR);

CALL JOURNAL(text);
```

where "text" is a string representing the log message.

In general, messages are of three different types: request, error, and completion. Request messages merely echo the requested service and the actual parameters passed to PLIDAIR. Error messages log abnormal conditions and error codes. Completion messages, on the other hand, detail completion codes following termination of PLIDAIR requests.

Many of these messages include information that is primarily useful to a programmer trying to isolate malfunctions of PLIDAIR itself. The casual user is encouraged to ignore those data that seem irrelevant to his problem. Below is a summary of the various journal log messages currently supported:

* VOLUME vvvvvv NOT MOUNTED

  This message may be generated by CREATE, CREALL, DELINDX, DELETE and RENAME upon detecting that the required control volume "vvvvvv" is not online.

* CREATE(dsname),MEMBER(mname),VOL(ser),
  SPACE(pri,sec,dir),RECFM(recfm),LRECL(lrecl),
  BLKSI(blksi),KEYLN(keyln),OPTCD(optcd)
  OPTCD(optcd),FILE(ddname)

  This message details the service request for CREATE and CREALL. The MEMBER and FILE clauses are only meaningful for CREALL.

* ALLOC(dsname),MEMBER(mname),FILE(ddname),OLD/SHR

This message records either an ALLOC or ALLOCK request.

* FREEDDN(ddname)

This message signals a FREEDDN file call.

* FREEDSN(dsname)

This message details a FREEDSN call.

* DELETE(dsname),MEMBER(mname)

This message occurs when deleting data sets or PDS-members.

* FINDDSN F(ddname)

This message is produced when trying to find the current allocation of ddname.

* FINDDSN F(ddname),DA(dsname),MEMBER(mname)

Details the results of a FINDDSN call.

* DELINDX(index)

This message signals deletion of index level "index".

* RENAME OLDDSN(olddsn),NEWDSN(newdsn),OLDMNM(oldname),
NEWMNM(newmname)

This message indicates an attempt to rename the given data sets or PDS-members.

* UNABLE TO RENAME ddn TRIED TO RENAME BACK

This message indicates that some abnormal condition arose impeding the completion of a rename operation. The old data set name remains unchanged; ddn is the system assigned ddn name currently allocated to the dsn to be renamed.

* OLD MEMBER(mname) UNAVAILABLE

This message indicates that a PDS-member rename operation cannot proceed because the OLD MEMBER name does not exist to begin with.

* NEW MEMBER(mname) IS DUPLICATE

Similarly when the new PDS-member already exists the rename operation cannot proceed.

* UNABLE TO OPEN DSN(dsn)

This message may be generated by ALLOCK, and both DELETE and RENAME of PDS members upon detection of some error condition that prevents the succesful opening of the dataset named by dsn.

* MISSING OR BAD PARM

This message may be generated by FINDDSN, DELINDX, and RENAME when one or more of their input parameters is improperly specified.

* UNABLE TO LOCATE dsn

This message is generated by FINDDSN when the named dataset dsn is not currently allocated.

* DAIR(callcode),RC(dairrc),CAMRC(catalogrc),
DALRC(dynallocrc),PLIRC(plidairrc)

This message occurs generally once per PLIDAIR call; it details completion of the requested service. The meanings of the terms are:

"callcode" is the generic DAIR opcode for the requested service. Its possible values are:

8 -- create and/or allocate a data set
24 -- delete and/or free a data set
28 -- allocate a ddn to the terminal
52 -- build or delete an attribute control block

"dairrc" is the return code from the DAIR function. For its values, see reference [1].

"catalogrc" is the return code from catalog management. For its values, see reference [1].

"dynallocrc" is the return code from dynamic allocation. For its values, see reference [1].

"plidairrc" is the return code from PLIDAIR to its caller. It is zero for a successful call. For its exceptional values, see the section named PARAMETERS.

* PDS(pdscode),PDSRC(pdsrc),RBYTE(rbyte),PLIRC(plirc)

This message occurs once per PDS management call: it details the completion of the service generically identified by pdscode. The meanings of the terms are:

"pdscode" identifies the operation [ref. 5] being performed. Its possible values are:

10 -- stowa operation
11 -- stowr operation
12 -- stowd operation
13 -- stowc operation
20 -- bldl operation

"pdsrc" is the return code from the operation. For its values, see STOW and BLDL in reference [5].

"rbyte" is the "r" part of "ttr". It is sometimes used to signal error conditions. See STOW and BLDL in reference [5].

"plidairrc" is the return code from PLIDAIR to its caller. It is zero for a successful call. For its exceptional values, see the section named PARAMETERS.

* CATALOG(catcode),CAMLST(op1,op2,op3,op4),CAMRC(camrc),
INDXS(indxs),LOCATE(locate),SCRATCH(scratch),PLIRC(plirc)

This message occurs once per catalog management call; it details completion of the indicated service. The meanings of the terms are:

"catcode" identifies the operation being performed [ref. 5]. Its possible values are:

0 -- index operation
1 -- scratch operartion
2 -- rename operation
3 -- catalog operation

"op1...op4" are the option bytes of the CAMLST macro-instruction. Their values are meaningful only through direct reference to the PLIDAIR source listing.

"indxs" is the number of index levels searched, as defined in reference [2].

"locate" is the return code from the LOCATE service, as defined in reference [2].

"scratch" is the return code from the SCRATCH service, as defined in reference [2].

"plidairrc" is the return code from PLIDAIR to its caller. It is zero for a successful call. For its exceptional values, see the section named PARAMETERS.

REFERENCES

[1] IBM Corporation, "IBM System/360 Operating System Time Sharing Option -- Guide to Writing a Terminal Monitor Program or a Command Processor." IBM order no. GC28-6764-2, 1972.

[2] IBM Corporation, "OS Data Management for System Programmers." IBM order no. GC28-6550-11, 1973.

[3] Ludlam, "CCNDEVTP Macro." UCLA document S-169, January, 1975.

[4] IBM Corporation, "OS PL/I Optimizing Compiler: Programmer's Guide." IBM order no. SC33-0006-4, 1976.

[5] IBM Corporation, "OS Data Management Macro Instructions." IBM order no. GC26-3794-1, 1973.

[6] IBM Corporation, "IBM System/360 Operating System Time Sharing Option -- Command Language Reference." IBM order no. GC28-6732-4, 1973.

PART VII

TXSTAX -- ATTENTION HANDLING IN PL/I

This section is separately available
as UCLA document UCNSW-408

7. PART VII: THE TXSTAX PACKAGE

7.1. FUNCTIONS OF THE TXSTAX PACKAGE

TXSTAX is a package of PL/I-callable (IBM Optimizing Compiler) entries for managing TSO attention interruptions. It antedates the IBM-provided attention support, and differs from it in that:

* ON-units are not used ... the program detects attentions by testing and/or waiting on an ECB.

* The compiler need not generate extra code for testing for attentions within every statement.

* Several different flavors of attention handling are provided.

* Multiple levels of attention handling are supported.

* The program can enable and disable the entire attention handling system.

7.2. ENTRIES

There are five entries to the TXSTAX package. Only two of these take parameters. This section sumarizes functions; parameters are described under "Syntax of TXSTAX/TXSTAXA."

7.2.1. TXSTAX

The TXSTAX entry sets up an attention-handling element and places it on top of the system's attention-handling stack for this job. If the top element of the stack belongs to this task, it is replaced. Otherwise, the stack is pushed down. Parameters to this entry (described later) determine the method of attention handling associated with the element. The element remains in effect until either:

* It is replaced by another TXSTAX call;

* It is cancelled by a TXSTAXC call;

* It is dequeued by termination of this task.

7.2.2. TXSTAXA

The TXSTAXA entry sets up an attention-handling element and places it on top of the system's attention-handling stack for this job. All existing stack elements are pushed down, regardless of their task association. Parameters to this entry (described later) determine the method of attention handling associated with the element. The element remains in effect until either:

* It is replaced by a TXSTAX call;

* It is cancelled by a TXSTAXC call;

* It is dequeued by termination of this task.

7.2.3. TXSTAXC

The TXSTAXC entry pops up the stack of attention-handling elements, if the top element belongs to this task. There are no parameters to this entry.

7.2.4. TXSTAXN

The TXSTAXN entry disables all attention processing for the entire job. This entry should be used with extreme care, and only by programmers thoroughly familiar with the cautions documented uder the DEFER=YES option of the STAX macro-instruction. There are no parameters to this entry.

7.2.5. TXSTAXY

The TXSTAXY entry enables attention processing if it has been disabled through TXSTAXN. Any pending attentions will then be processed "simultaneously" (see the sections named "Limitations"). There are no parameters to this entry.

7.3. METHODS OF ATTENTION HANDLING

Exactly how attentions are handled depends on the parameters passed to the TXSTAX or TXSTAXA routine. Of the many options available to the Assembler-language programmer, only a few are presently available through this package; however, further optional parameters may be defined in the future.

7.3.1. SYNTAX OF TXSTAX/TXSTAXA

An attention-processing element is created by a call in one of the forms:

```
(opt. 1):  CALL entry (ecb, area);
(opt. 2):  CALL entry (ecb, area, prompt);
(opt. 3):  CALL entry (ecb, area, prompt, buffer);
```

where:

* "entry" is either "TXSTAX" or "TXSTAXA", depending on whether you wish to replace an existing stack element.

* "ecb" is FIXED BINARY (31), and has been previously set to zero. The ECB is posted (with a post code of 255) whenever an attention is serviced by this element. The caller may WAIT on the ECB and/or test its completion bit and/or post code.

* "area" is the first word of a 7-fullword connected area. The caller should not modify or free this area so long as the attention-handling element remains pending. Notice that, while this datum must be declared as an array or structure, it must be passed as a scalar (i. e., pass the first element). If option 1 or 3 is being used, this area need be only 6 fullwords long.

* "prompt" is an optional CHAR (*) parameter to be used as a prompt string. It may be fixed or VARYING ALIGNED.

* "buffer" is an optional CHAR (*) VARYING ALIGNED string which is to receive the input string from the terminal user.

7.3.2. BEHAVIOR OF OPTION 1

If option 1 is coded, the created element specifies attention handling without input or output. When an attention occurs, the ECB is POSTed with post code 255, and processing continues. Program execution is effectively uninterrupted. The user is not prompted for input, and has no opportunity to enter a second attention. He is thus virtually prevented from attaining any higher level of control (as the TEST or READY levels). For this reason, this option should be used with caution, and never in an un-debugged program.

See the sections named "Limitations" for situations in which option 1 may behave otherwise.

7.3.3. BEHAVIOR OF OPTION 2

If option 2 is coded, the created element specifies attention handling with program execution suspended. When an attention occurs, the running program is stopped, and the user is prompted for input with the character string in the position named "prompt." These cases exist:

* If the user enters only a carriage return, the interruption is cancelled. The ECB is not posted, and program execution is resumed.

* If the user enters a second attention, this stack element is bypassed, and the interruption is processed according to the next element on the stack. If and when that element completes normally, program execution is resumed. The ECB is not posted.

* If the user enters anything else, the ECB is posted and program execution resumes. Note that the string entered by the user is not made available to the program. No processing of any kind is done on the string -- e.g., "?" is not considered a request for secondary messages.

7.3.4. BEHAVIOR OF OPTION 3

If option 3 is coded, the created element specifies attention handling with program execution suspended. When an attention occurs, the running program is stopped, and the user is prompted for input with the character string in the position named "prompt." These cases exist:

* If the user enters only a carriage return, the interruption is cancelled. The ECB is not posted, and program execution is resumed.

* If the user enters a second attention, this stack element is bypassed, and the interruption is processed according to the next element on the stack. If and when that element completes normally, program execution is resumed. The ECB is not posted.

* If the user enters anything else, the ECB is posted, the string entered is placed into "buffer", the current length field of that variable is set accordingly, and program execution then resumes. No processing of any kind is done on the string -- e.g., "-" is not considered a continuation indicator, and "?" is not considered a request for secondary messages.

7.4. LIMITIATIONS OF MVT/TSO

Attention handling under MVT/TSO is affected by accidents of timing in a way that makes it somewhat less than deterministic. Since the only way to enter multiple attentions from a terminal is to enter a sequence of single attentions, this document has described the handling of "sequential" interruptions. However, under certain circumstances, such attentions are processed as "simultaneous" interruptions. This will be the case if the job is operating with attention handling disabled (as after a call to TXSTAXN), or if the job remains swapped out over the entire interval during which the attentions are entered.

When multiple interruptions are processed simultaneously, a single element of the attention-handling stack is selected, that being the one whose depth in the stack matches the number of interruptions being processed (or the bottom one, if the stack depth is exceeded). Only this element is used; if and when it completes normally, program execution is resumed. No elements above the selected one have any effect on the processing of the attentions.

7.5. LIMITATIONS OF THIS PACKAGE

Some capabilities available to the Assembler programmer which are not currently supported by this package include:

* Rescheduling -- There is a type of attention-handling element which, if interrupted by a second sequential attention, will again receive control after the second interruption has been processed. This type of element is not available from this package.

* Logical Line Processing -- There is a type of attention-handling element which returns a logical line, rather than a physical line, of user input. This mostly means that system code processes hyphens and question marks when they appear in the positions where they are defined to have special significance. Such processing is not available from this package.